Reviewing Coverage Information

This topic explains how to review coverage information from tests run with C++test. Coverage can be tracked for unit tests and manual or automated tests run at the application level.

Reviewing coverage statistics helps you measure the coverage of your current test suite and decide what additional test cases should be added. C++test can report a variety of code coverage types, including function, line, path, basic block, decision (branch), simple condition, and MCDC coverage metrics.

Sections include:

Analyzing Coverage in the GUI

Coverage Summary

To view a summary of coverage information after C++test executes unit test cases:

1. Open the Coverage view.
   - If it is not available, choose **Parasoft> Show View> Coverage**.
2. From the pull-down menu in the Coverage view’s toolbar, choose **Type> [Desired_Coverage_Type].**
   - See Understanding Coverage Types for a description of the available coverage types.

Coverage statistics for the project and all analyzed files and functions will be displayed in the Coverage view. If no coverage is shown after test execution, check that 1) coverage was enabled and 2) you are displaying the type of coverage you configured C++test to calculate. The view’s toolbar indicates the coverage metric being displayed.

You can sort and search through the results using the available GUI controls.

Annotated Source Code

When a tested file is opened in the editor, C++test will use green highlights to indicate which code was cover ed and pink highlights to indicate which code was not covered. Lines that are not executable will not be highlighted.

Note that you need to double-click on a function to start off path coverage view.

Note

There is a limit for the number of coverage elements (paths, blocks etc.) that can be computed by C++test. When the actual number of elements on a given level (function, file or a project) exceeds the limit, which is 2147483647, C++test will display "N/A" in the report for a given element. In such cases, the Coverage view will contain appropriate messages, e.g. [no paths], [no blocks].

Back Trace from Coverage Elements to Test Cases

Knowing which test cases are related to each coverage element can help you better assess how to extend test cases to improve coverage.

To see which test cases are related to a coverage element:

1. Select that coverage item in the editor.
   - C++test will consider the current selection / cursor position, not the mouse pointer location.
2. Right-click that selection, then choose **Parasoft> C++test> Show test case(s) for covered element**. All related test cases will be highlighted in the Test Case Explorer.

Coverage Data for Specific Test Cases

To analyze the coverage for individual test cases:

1. In the Test Case Explorer, select the test case(s) whose coverage you want to analyze.
    - The Test Case Explorer opens by default. If it is not available, you can open it by choosing **Parasoft> Show View> Test Case Explorer**. For details on understanding and navigating the Test Case Explorer, see Exploring the C++test UI.
2. In the Coverage view, click the **Synchronize with selection in Test Case Explorer** filter (a double cog-wheel button).

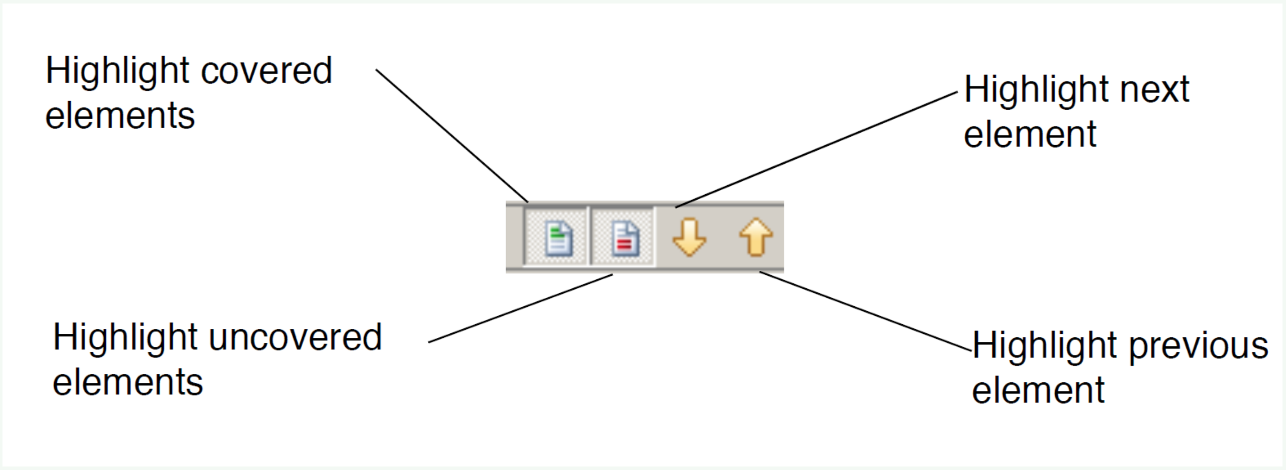
Coverage statistics and coverage highlights (in the code editor) will be computed/presented for the selected test cases only.

Tip: Using Toolbar Buttons and Menus to Explore Coverage

The Coverage view provides several buttons and menu commands to help you explore the coverage details reported.

The buttons on the right of the toolbar allow you to collapse, delete, search for results, and synchronize with the current selection in the Test Case Explorer.

In addition, the following toolbar buttons allow you to show/hide covered elements, show/hide uncovered elements, highlight the next path (for path coverage only), and highlight the previous path (for path coverage only).


Highlight covered elements
Highlight next element
Highlight uncovered elements
Highlight previous element

The drop-down menu provides commands that allow you to sort results by ascending/descending name or coverage, as well as to select the desired coverage type.

Note: Coverage View Results Upon Startup

To optimize performance, the Coverage view starts in a uninitialized mode—displaying only project-level coverage summaries. This mode is indicated with grayed project icons:

Coverage in this mode may not be current—for instance, if the source code was modified outside of this IDE. You can update the coverage view by simply expanding a project node. Or, you can wait for the view to be automatically updated during execution.

Analyzing Coverage in Reports

For details on configuring, generating, and reviewing reports, see Reviewing Results.

Understanding Coverage Types

C++test supports the following coverage types:

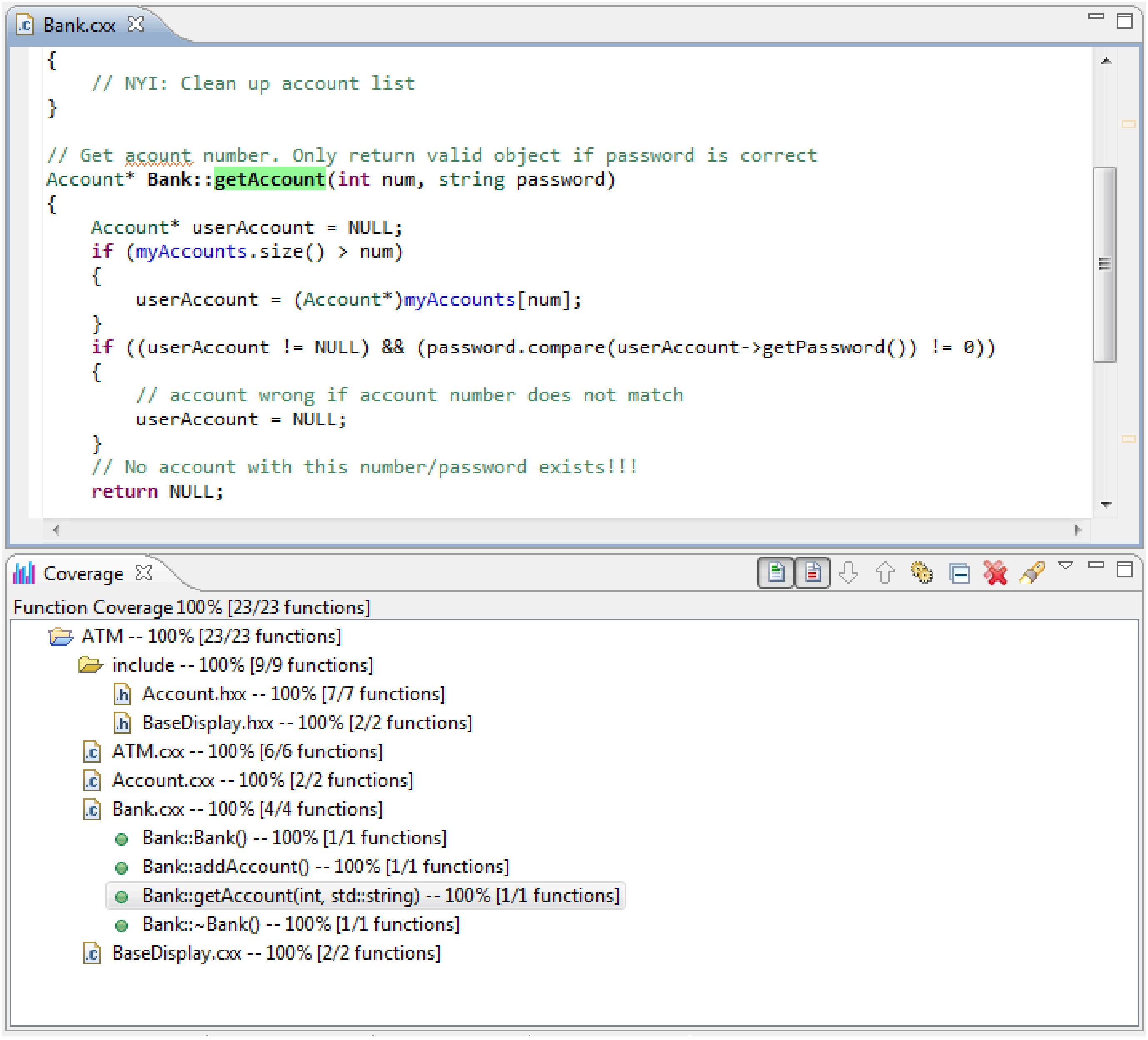
- Line Coverage
- Statement Coverage
- Block Coverage
- Path Coverage
- Decision (Branch) Coverage
- Modified Condition/Decision Coverage (MC/DC)
- Simple Condition Coverage
- Function Coverage
- Call Coverage

To help understand how these coverage types are handled by C++test, be sure to read and understand the terms in the following table:

Concept	Description
Basic Block	A sequence of non-branching statements; a linear sequence of code with no control flow route branchings.
Path	A unique sequence of basic blocks starting from the function entry to the point of exit.
Decision/Branch	Decision/Branch is the possible control flow decision to be taken at the branching point in the code. C++test considers if-else, for, while, do-while, and switch instructions as the branching points. C++test does not take into account such dynamic branching points as exception handlers (throw-catch statements).
Boolean expression	In C++, a boolean expression is simply an expression that has a 'bool' type. In C, C++test treats the following as boolean expressions: • A relational operator (<, <=, ==, >, >=, !=) with non boolean arguments. • Each argument of boolean operator (||, &&, !). • Condition in if, for, while and do-while instructions. • Condition in ? operator.
MC/DC Decision	A top-level boolean expression composed of conditions and zero or more boolean operators. C++test computes MC/DC and SCC on all boolean non-constant expressions in the source code except constructor initializers and function default arguments.
Condition	An atomic boolean non-constant expression that is a part of the MC /DC decision. A sub-expression of the MC/DC decision is considered to be a condition if it does not contain boolean operators (&&, ||, !). If a given atomic expression appears more than once in a decision, each occurrence is a distinct condition.

Function Coverage

Indicates how many functions in the source code were reached at least once during execution. Complete, 100% function coverage is obtained if all functions are reached at least once.



```
{
    // NYI: Clean up account list
}

// Get acount number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPassword()) != 0))
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
```

```
Function Coverage 100% [23/23 functions]
  ATM -- 100% [23/23 functions]
    include -- 100% [9/9 functions]
      Account.hxx -- 100% [7/7 functions]
      BaseDisplay.hxx -- 100% [2/2 functions]
    ATM.cxx -- 100% [6/6 functions]
    Account.cxx -- 100% [2/2 functions]
    Bank.cxx -- 100% [4/4 functions]
      Bank::Bank() -- 100% [1/1 functions]
      Bank::addAccount() -- 100% [1/1 functions]
      Bank::getAccount(int, std::string) -- 100% [1/1 functions]
      Bank::~Bank() -- 100% [1/1 functions]
    BaseDisplay.cxx -- 100% [2/2 functions]
```

Call Coverage

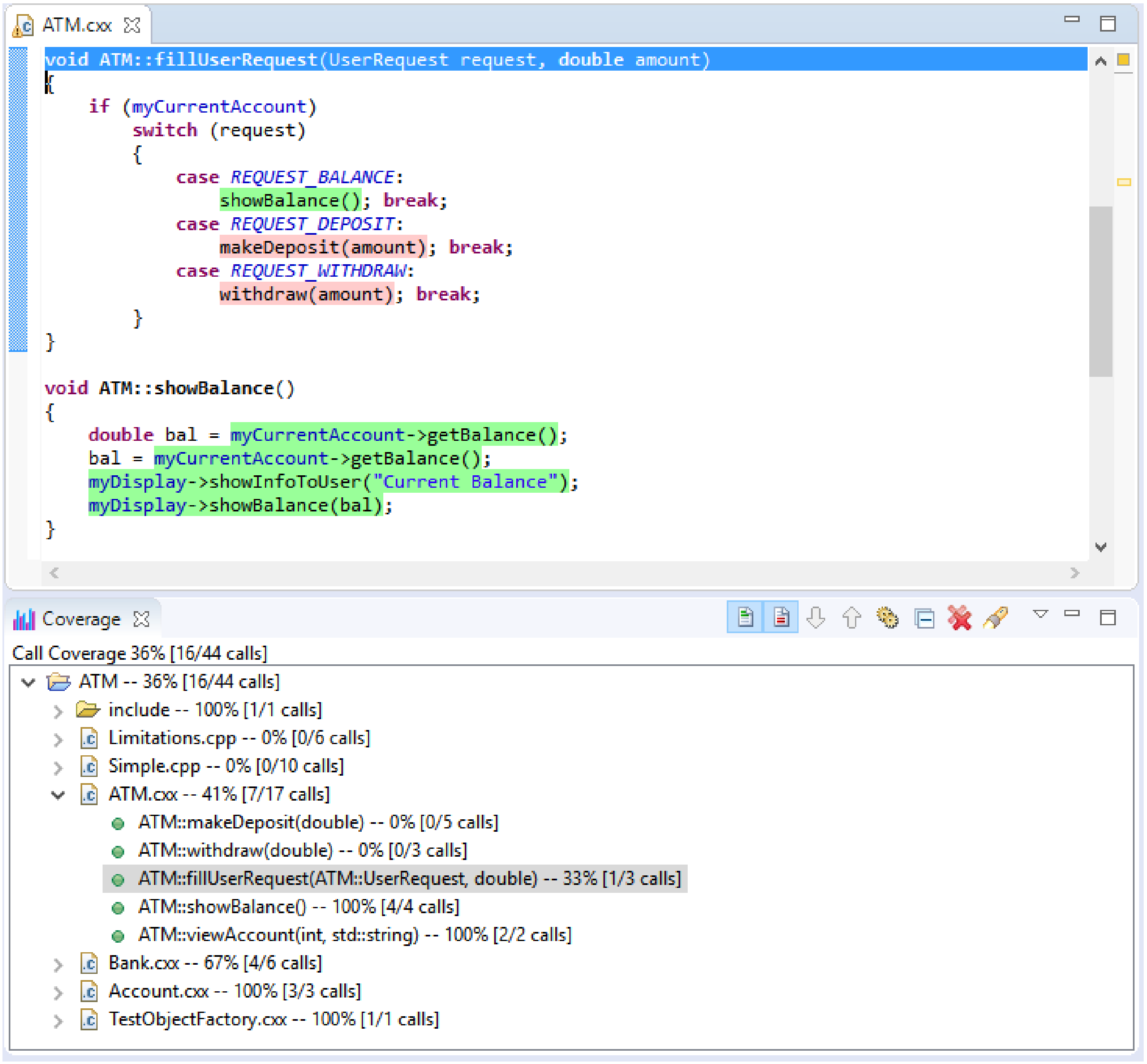
Indicates how many defined function or method calls were executed at program runtime. Complete, 100% call coverage is obtained if all functions or methods calls were executed.

Limitations:

- Constructors calls, both explicit and implicit, are not included in call coverage calculation.
- Implicit destructor calls are not included in call coverage calculation.
- C++ operators, both predefined and overloaded, are not included in call coverage calculation.

Example:

```
class A
{
public:
    A() {}
    A(const A&) {}
    A(int val) {}
    A& foo()
    {
        return *this;
    }
    ~A() {}
};


void funcA(const A& a) {}
A& operator+(const A & a, const A & b) {}


void defaultConstructorCall()
{
    A objA;                // Default constructor call - not included
    A objArr[10];          // Default constructor calls - not included
    A objB = objA;         // Copy constructor call       - not included
    A objC = 10;           // Implicit call of A(int) constructor - not
included

    funcA(10);             // Implicit call of A(int) constructor - not
included

    A res = objA + objA;   // Operator + call - not included
                                                   // Copy constructor
call - not included

    A* objD = new A();    // Default constructor call - not included
                                                  // Operator new call -
not included

    delete objD;          // Destructor call - not included
                                                  // Operator new call -
not included

    A* objE = new A[10];  // Default constructor calls  - not included
                          // Operator new call - not included

    delete[] objE;        // Destructor calls - not included
                          // Implicit destructor calls - not included
}
```



ATM.cxx

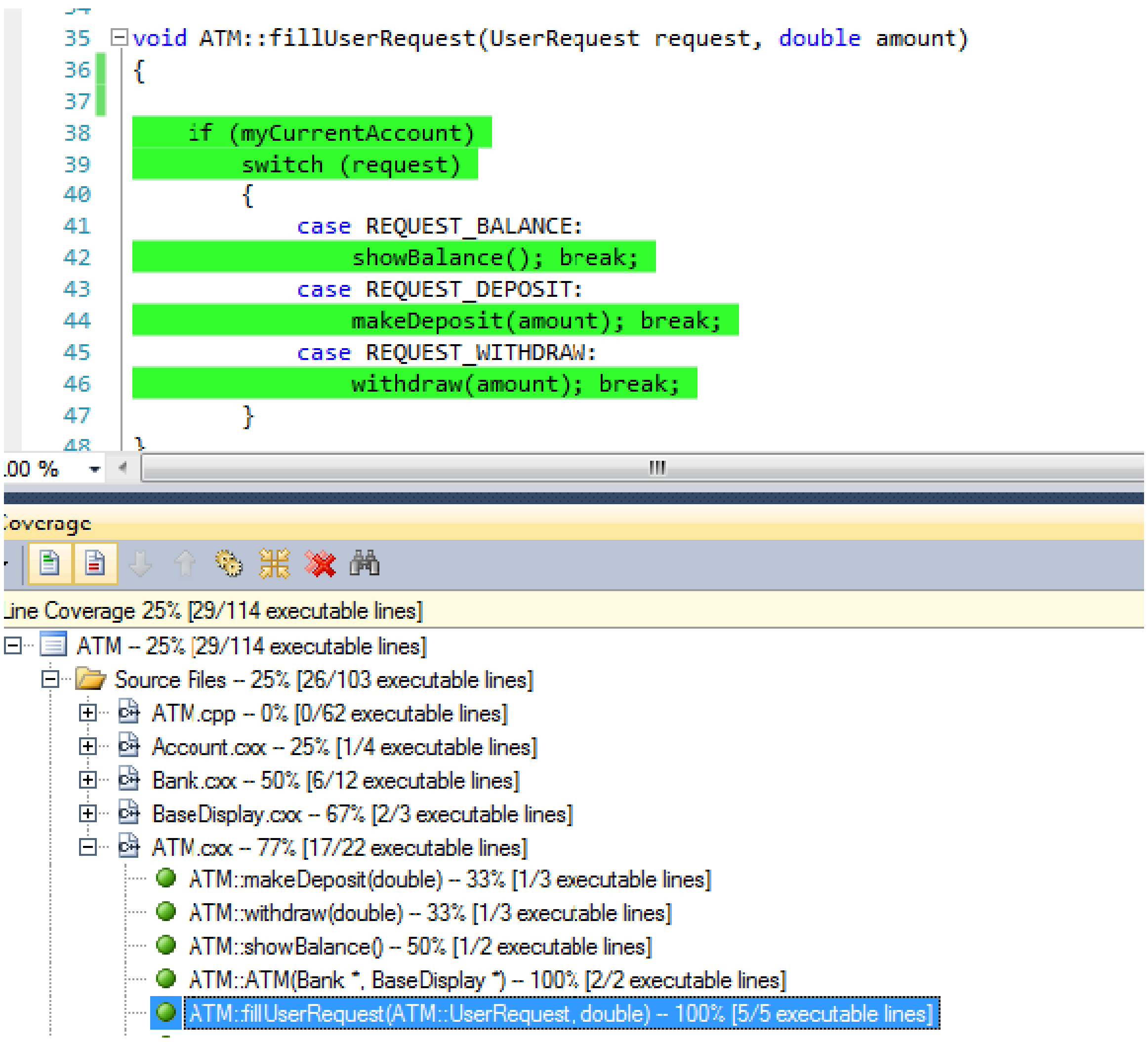
```
void ATM::fillUserRequest(UserRequest request, double amount)
{
    if (myCurrentAccount)
        switch (request)
        {
            case REQUEST_BALANCE:
                showBalance(); break;
            case REQUEST_DEPOSIT:
                makeDeposit(amount); break;
            case REQUEST_WITHDRAW:
                withdraw(amount); break;
        }
}

void ATM::showBalance()
{
    double bal = myCurrentAccount->getBalance();
    bal = myCurrentAccount->getBalance();
    myDisplay->showInfoToUser("Current Balance");
    myDisplay->showBalance(bal);
}
```

Coverage

```
Call Coverage 36% [16/44 calls]
ATM -- 36% [16/44 calls]
  include -- 100% [1/1 calls]
  Limitations.cpp -- 0% [0/6 calls]
  Simple.cpp -- 0% [0/10 calls]
  ATM.cxx -- 41% [7/17 calls]
    ATM::makeDeposit(double) -- 0% [0/5 calls]
    ATM::withdraw(double) -- 0% [0/3 calls]
    ATM::fillUserRequest(ATM::UserRequest, double) -- 33% [1/3 calls]
    ATM::showBalance() -- 100% [4/4 calls]
    ATM::viewAccount(int, std::string) -- 100% [2/2 calls]
  Bank.cxx -- 67% [4/6 calls]
  Account.cxx -- 100% [3/3 calls]
  TestObjectFactory.cxx -- 100% [1/1 calls]
```

Line Coverage

Indicates how many executable lines of source code were reached by the control flow at least once. Complete, 100% line coverage is obtained if all executable lines are reached at least once.

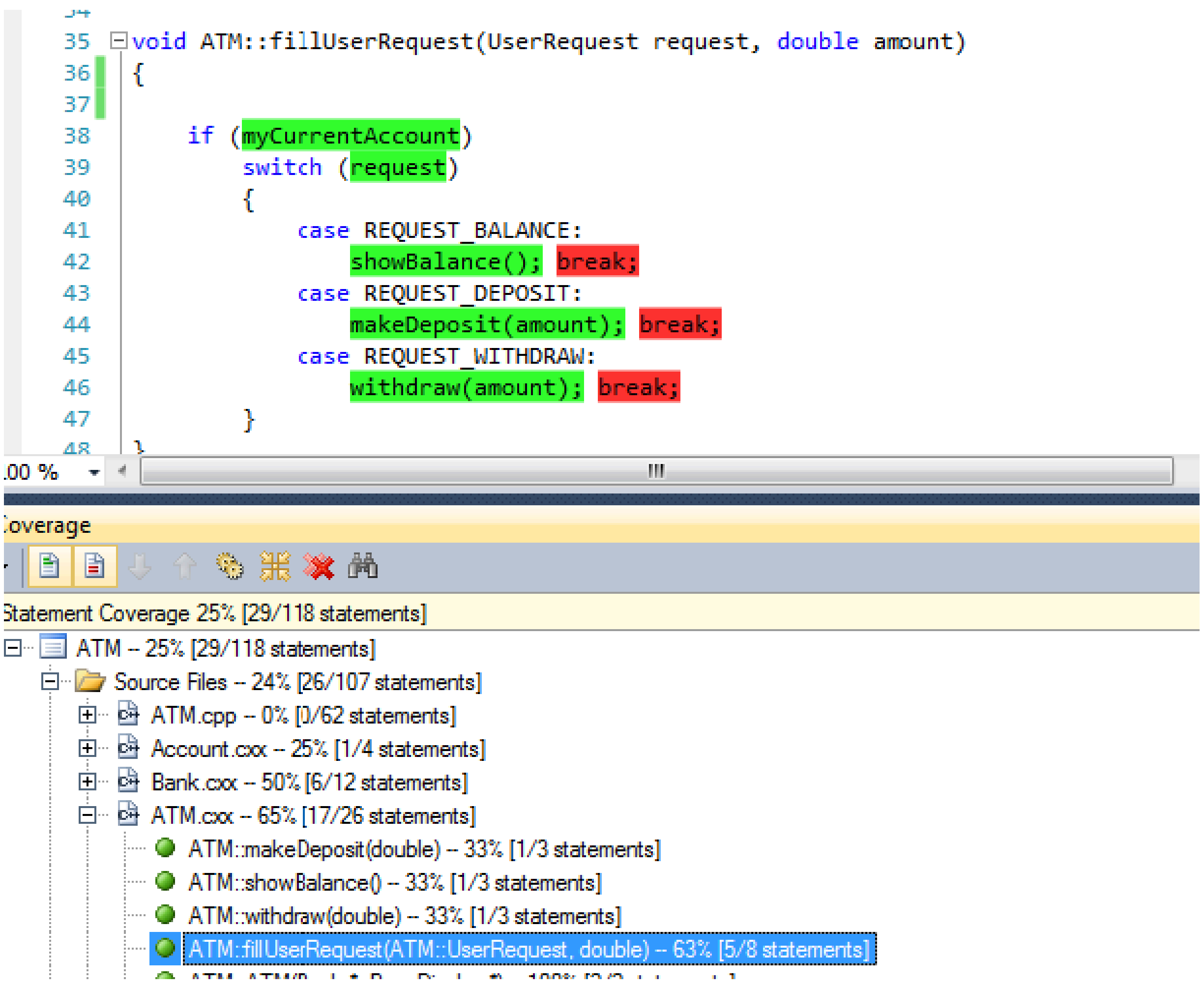


```
void ATM::fillUserRequest(UserRequest request, double amount)
{

    if (myCurrentAccount)
        switch (request)
        {
            case REQUEST_BALANCE:
                showBalance(); break;
            case REQUEST_DEPOSIT:
                makeDeposit(amount); break;
            case REQUEST_WITHDRAW:
                withdraw(amount); break;
        }
}
```

```
Line Coverage 25% [29/114 executable lines]
ATM -- 25% [29/114 executable lines]
    Source Files -- 25% [26/103 executable lines]
        ATM.cpp -- 0% [0/62 executable lines]
        Account.cxx -- 25% [1/4 executable lines]
        Bank.cxx -- 50% [6/12 executable lines]
        BaseDisplay.cxx -- 67% [2/3 executable lines]
        ATM.cxx -- 77% [17/22 executable lines]
            ATM::makeDeposit(double) -- 33% [1/3 executable lines]
            ATM::withdraw(double) -- 33% [1/3 executable lines]
            ATM::showBalance() -- 50% [1/2 executable lines]
            ATM::ATM(Bank *, BaseDisplay *) -- 100% [2/2 executable lines]
            ATM::fillUserRequest(ATM::UserRequest, double) -- 100% [5/5 executable lines]
```

Statement Coverage

Indicates how many executable source code statements were reached by the control flow at least once. Complete, 100% statement coverage is obtained if all executable statements are reached at least once.

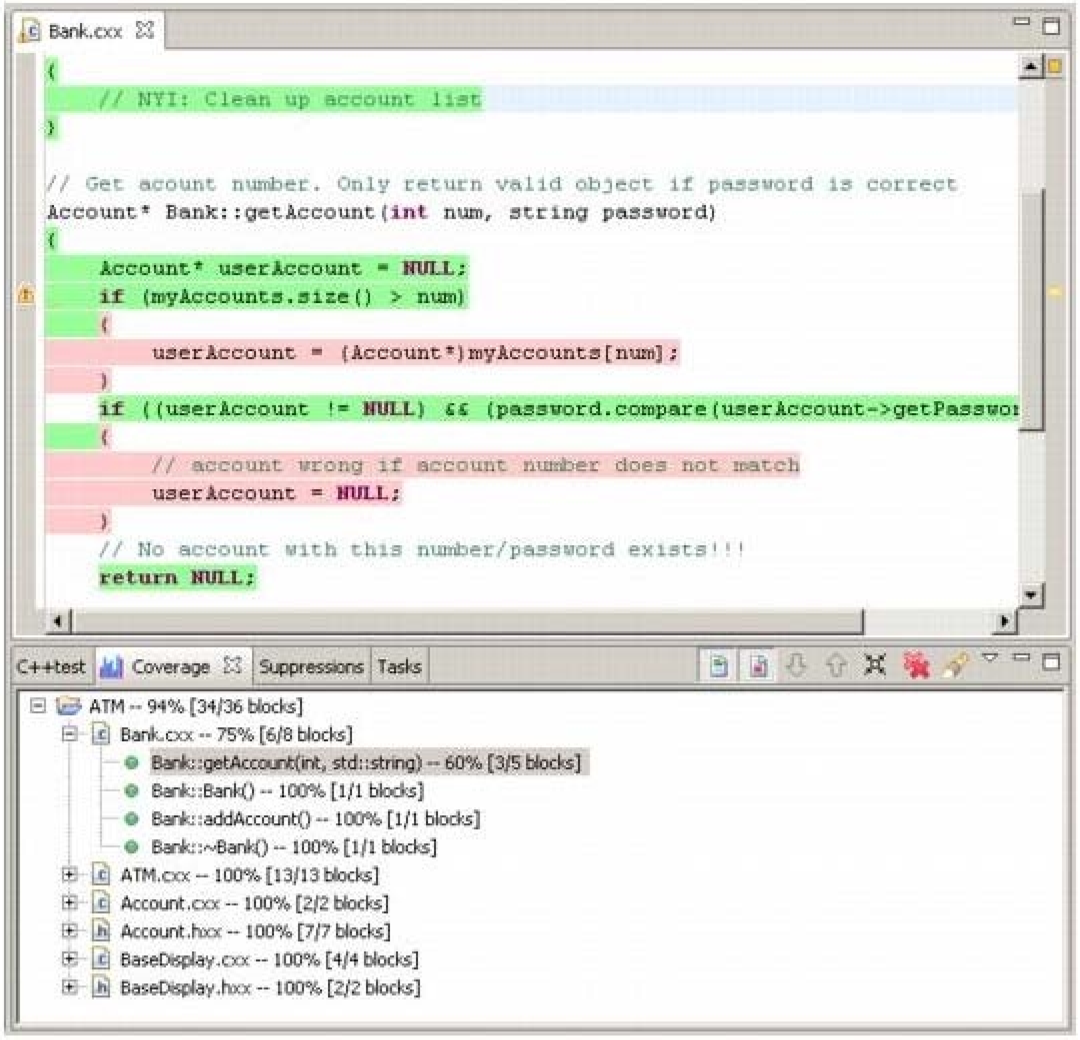


```
void ATM::fillUserRequest(UserRequest request, double amount)
{

    if (myCurrentAccount)
        switch (request)
        {
            case REQUEST_BALANCE:
                showBalance(); break;
            case REQUEST_DEPOSIT:
                makeDeposit(amount); break;
            case REQUEST_WITHDRAW:
                withdraw(amount); break;
        }
}
```

```
Statement Coverage 25% [29/118 statements]
ATM -- 25% [29/118 statements]
    Source Files -- 24% [26/107 statements]
        ATM.cpp -- 0% [0/62 statements]
        Account.cxx -- 25% [1/4 statements]
        Bank.cxx -- 50% [6/12 statements]
        ATM.cxx -- 65% [17/26 statements]
            ATM::makeDeposit(double) -- 33% [1/3 statements]
            ATM::showBalance() -- 33% [1/3 statements]
            ATM::withdraw(double) -- 33% [1/3 statements]
            ATM::fillUserRequest(ATM::UserRequest, double) -- 63% [5/8 statements]
```

Block Coverage

Similar to Line Coverage—except that with Block Coverage, the unit of measured code is a basic block (see the definition of this term in the previous table). This indicates how many basic blocks in the source code were reached by the control flow at least once.



```
{
    // NYI: Clean up account list
}

// Get acount number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPasswo
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
```

```
ATM -- 94% [34/36 blocks]
    Bank.cxx -- 75% [6/8 blocks]
        Bank::getAccount(int, std::string) -- 60% [3/5 blocks]
        Bank::Bank() -- 100% [1/1 blocks]
        Bank::addAccount() -- 100% [1/1 blocks]
        Bank::~Bank() -- 100% [1/1 blocks]
    ATM.cxx -- 100% [13/13 blocks]
    Account.cxx -- 100% [2/2 blocks]
    Account.hxx -- 100% [7/7 blocks]
    BaseDisplay.cxx -- 100% [4/4 blocks]
    BaseDisplay.hxx -- 100% [2/2 blocks]
```

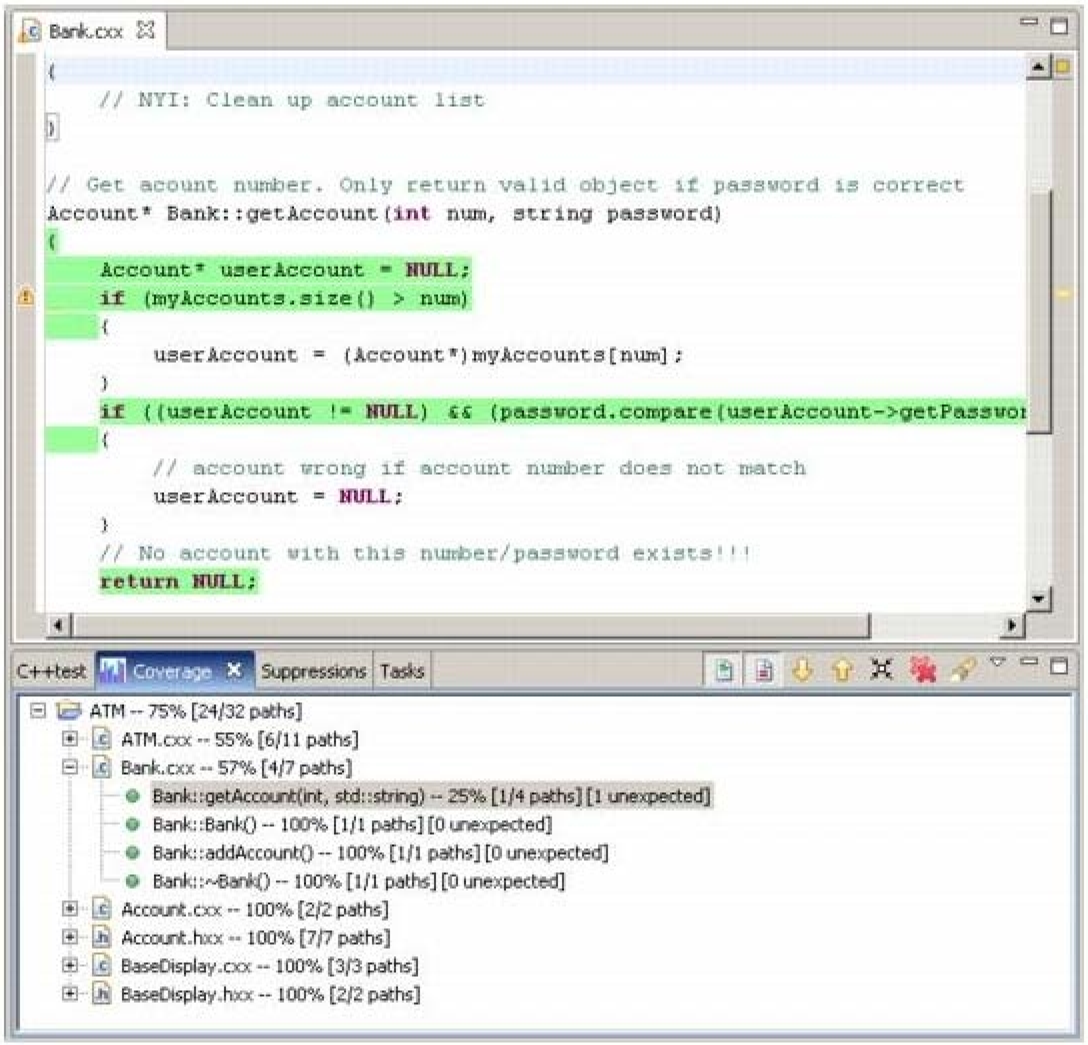
Path Coverage

Indicates if each possible path in a given function was followed by the control flow. Branching points used to single out paths (see an explanation of this term in the previous table) are the same as in Decision (Branch) Coverage.

Since loops introduce an unbounded number of paths, this measure considers only a limited number of looping possibilities. C++test considers two possibilities for while-loops and for-loops: zero and at least one repetition.

In the source editor, C++test highlights one path at a time. To view a path in the source code editor, double-click on the appropriate function node in the Coverage view. To navigate between paths for the function, use the **Highlight next element** and **Highlight previous element** buttons in the Coverage view toolbar.

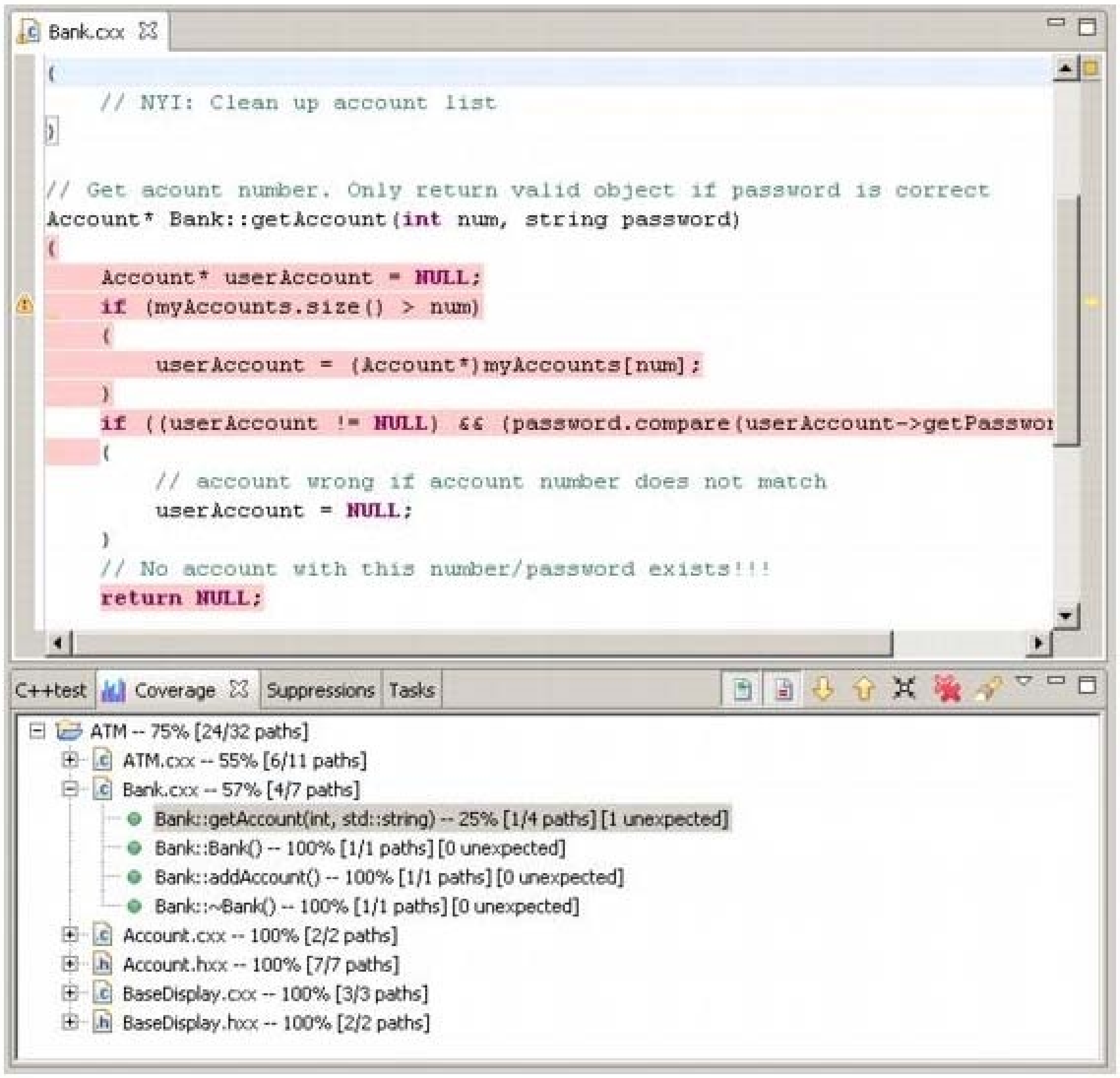
To prompt C++test to show only uncovered paths, disable the **Highlight covered elements** button in the Coverage view toolbar. To prompt C++test to show only covered paths, disable the **Highlight not covered elements**. Note that the **Highlight next/prev element** buttons iterate through unexpected paths only when the **Highlight not covered elements** button is disabled.



Bank.cxx

```
{
    // NYI: Clean up account list
}

// Get acount number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPasswo
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
```

```
ATM -- 75% [24/32 paths]
    ATM.cxx -- 55% [6/11 paths]
    Bank.cxx -- 57% [4/7 paths]
        Bank::getAccount(int, std::string) -- 25% [1/4 paths] [1 unexpected]
        Bank::Bank() -- 100% [1/1 paths] [0 unexpected]
        Bank::addAccount() -- 100% [1/1 paths] [0 unexpected]
        Bank::~Bank() -- 100% [1/1 paths] [0 unexpected]
    Account.cxx -- 100% [2/2 paths]
    Account.hxx -- 100% [7/7 paths]
    BaseDisplay.cxx -- 100% [3/3 paths]
    BaseDisplay.hxx -- 100% [2/2 paths]
```



```
{
    // NYI: Clean up account list
}

// Get acount number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPasswo
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
```

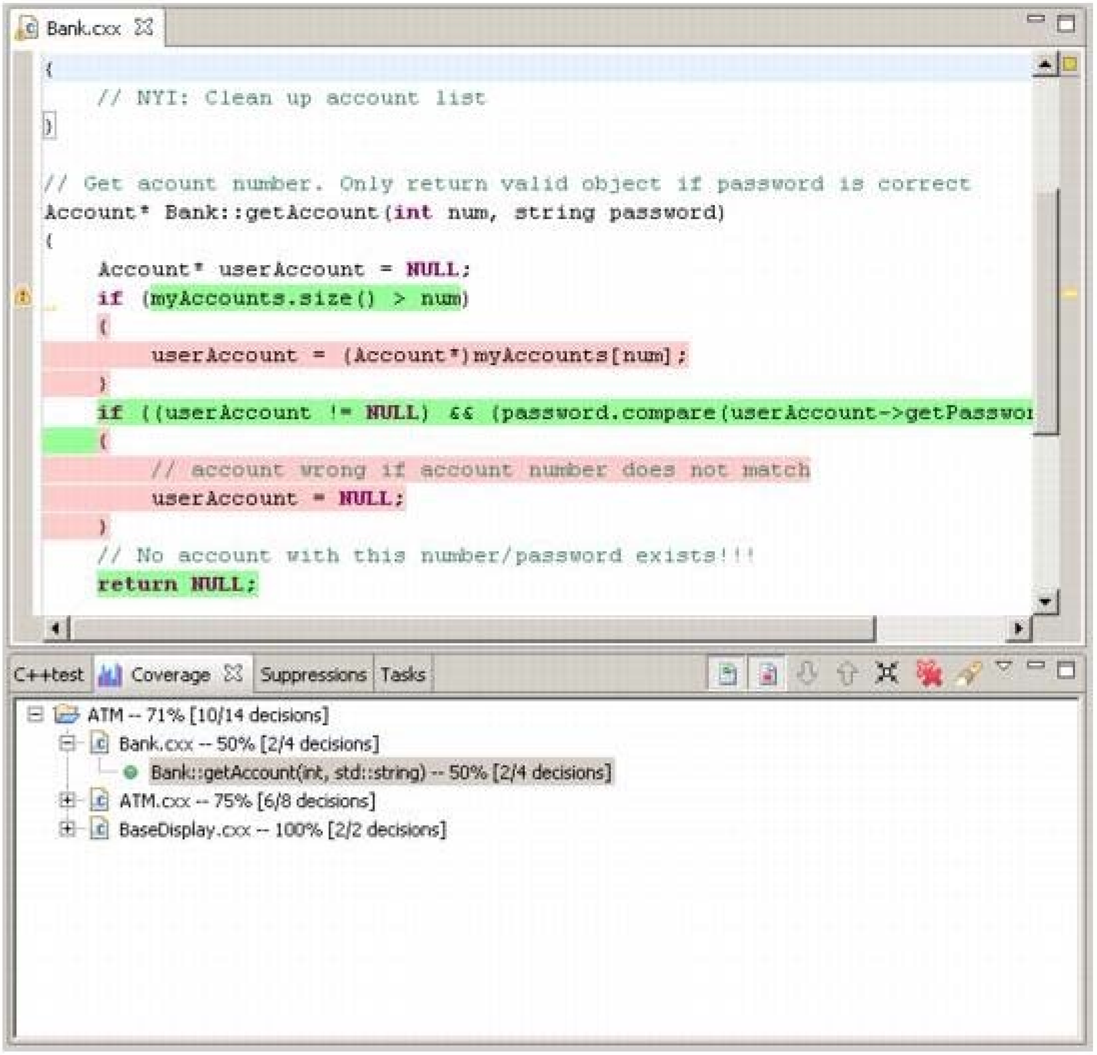
```
ATM -- 75% [24/32 paths]
    ATM.cxx -- 55% [6/11 paths]
    Bank.cxx -- 57% [4/7 paths]
        Bank::getAccount(int, std::string) -- 25% [1/4 paths] [1 unexpected]
        Bank::Bank() -- 100% [1/1 paths] [0 unexpected]
        Bank::addAccount() -- 100% [1/1 paths] [0 unexpected]
        Bank::~Bank() -- 100% [1/1 paths] [0 unexpected]
    Account.cxx -- 100% [2/2 paths]
    Account.hxx -- 100% [7/7 paths]
    BaseDisplay.cxx -- 100% [3/3 paths]
    BaseDisplay.hxx -- 100% [2/2 paths]
```

Decision (Branch) Coverage

Indicates how many branches in source code control flow passed through. Complete, 100% coverage is obtained when every decision at all branching points took all possible outcomes at least once.

C++test considers the following statement types branching points in source code: `if-else`, `for`, `while`, `do-while`, and `switch`. C++test does not take into account such dynamic branching points as exception handlers (`throw-catch` statements).

If there are no decisions in a file, C++test reports that this metric is not available (using an "N/A" label).



```
{
    // NYI: Clean up account list
}

// Get acount number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPasswo
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
```

```
ATM -- 71% [10/14 decisions]
    Bank.cxx -- 50% [2/4 decisions]
        Bank::getAccount(int, std::string) -- 50% [2/4 decisions]
    ATM.cxx -- 75% [6/8 decisions]
    BaseDisplay.cxx -- 100% [2/2 decisions]
```

Modified Condition/Decision Coverage (MC/DC)

MC/DC conforms to the international technical standard DO-178B (RTCA), which specifies the software certification criteria for mission-critical equipment and systems within the aviation industry. This includes real time embedded systems.

According to the DO-178B standard, the following three conditions must be satisfied to obtain complete (100%) MC/DC coverage:

1. Every decision has taken all possible outcomes at least once.
2. Every condition in a decision has taken all possible outcomes at least once.
3. Every condition in a decision has been shown to independently affect that decision's outcome.

Since C++test considers that every condition and decision can have only two outcomes for MC/DC coverage (true or false), it checks only for the third option (c) listed immediately above—since point (c) implies conditions (a) and (b). A condition is shown to independently affect the outcome of a decision by varying only that particular condition while holding fixed all other possible conditions. To test one given condition, C++test looks for test cases where:

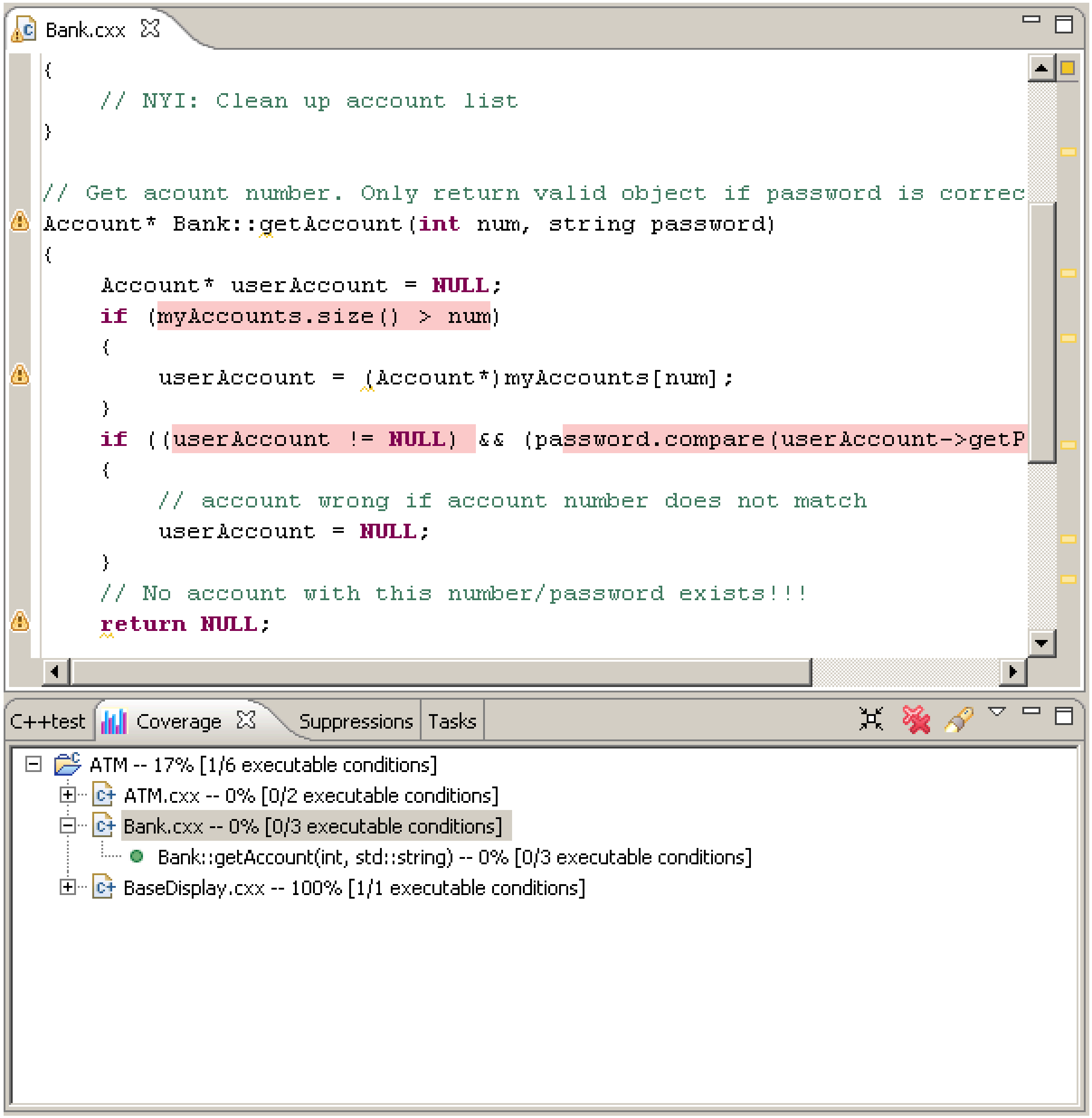
- The tested condition have both true and false outcomes.
- Other conditions in a decision do not change (or are not evaluated because operators in C/C++ are short-circuit logical).
- The outcome of a decision changes.

Thus, to calculate the MC/DC ratio, C++test uses the formula

```
MC/DC[%] = m/n
```

where `m` is the number of boolean conditions proven to independently affect a decision's outcome, and `n` is the total number of conditions in a decision.

Note that in order to make a single condition covered, at least two test cases need to be executed: one with the condition evaluated to `true` and a second with this condition evaluated to `false`.



```
{
    // NYI: Clean up account list
}

// Get acount number. Only return valid object if password is correc
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getP
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
```

```
ATM -- 17% [1/6 executable conditions]
    ATM.cxx -- 0% [0/2 executable conditions]
    Bank.cxx -- 0% [0/3 executable conditions]
        Bank::getAccount(int, std::string) -- 0% [0/3 executable conditions]
    BaseDisplay.cxx -- 100% [1/1 executable conditions]
```

MC/DC Example

For example, consider the following code:

```
    if (a && (b || c))
        // [...]
```

There are three simple conditions here: 'a', 'b' and 'c'. As a result, n (in the m/n MC/DC formula) will equal 3.

Now, let's assume that the following test cases were executed:

id	a	b	c	a && (b || c)

1	true	true	false	true
2	false	true	false	false
3	true	false	false	false

To compute MC/DC, C++test looks for the pairs of test cases where 1) the given decision was evaluated to a different value and 2) the value of only one condition changes (all other conditions remain unchanged).

In our example, there is one pair where independently changing the value of a affects the value of the complete decision:

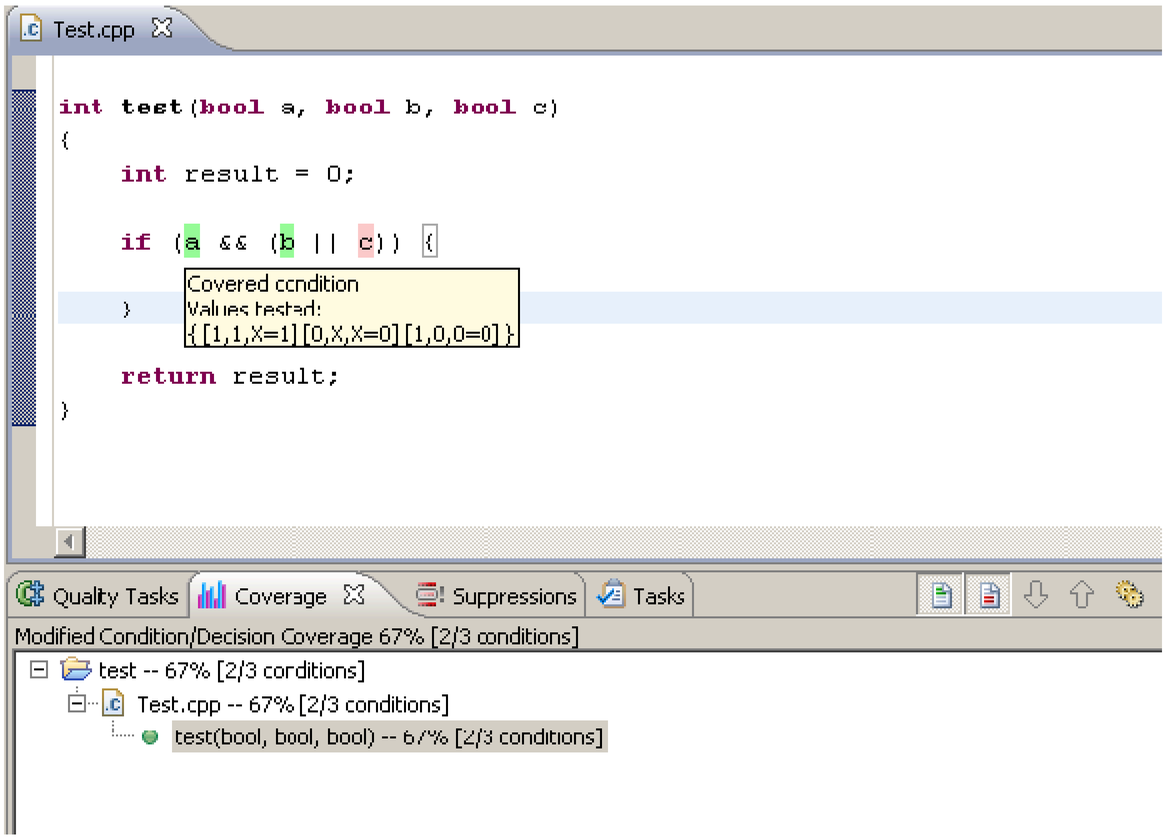
id	a	b	c	a && (b || c)
1	true	true	false	true
2	false	true	false	false

Another pair shows that independently changing the value of b affects the value of the complete decision:

id	a	b	c	a && (b || c)
1	true	true	false	true
3	true	false	false	false

This means that our test cases proved that a and b independently affect the value of the complete decision. In terms of MC/DC coverage, they are covered and the c condition is not covered.

C++test will report the MC/DC coverage for such an example to be 67% [2/3 conditions covered]. You can see the actual values that the conditions and decision evaluated to in a tool tip by placing your cursor above on the condition.



```
int test(bool a, bool b, bool c)
{
    int result = 0;

    if (a && (b || c)) {

    }

    return result;
}
```

Covered condition
Values tested:
{ [1,1,X=1] [0,X,X=0] [1,0,0=0] }

Modified Condition/Decision Coverage 67% [2/3 conditions]

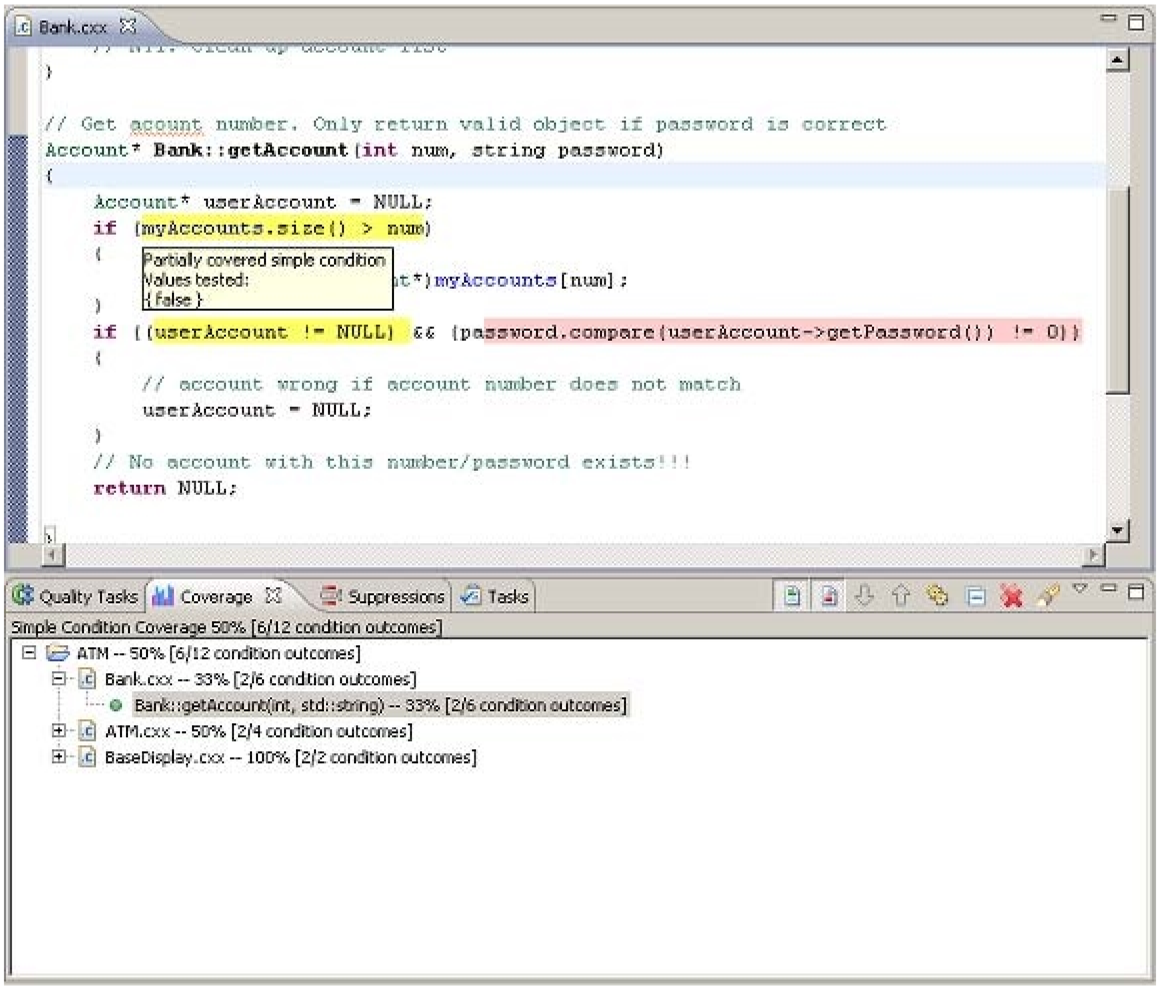
- test -- 67% [2/3 conditions]
  - Test.cpp -- 67% [2/3 conditions]
    - test(bool, bool, bool) -- 67% [2/3 conditions]

Simple Condition Coverage

Indicates the coverage for the outcomes of all decisions' conditions. The overall number of outcomes for a decision equals 2 * n, where n is the number of conditions in a decision. Therefore, to obtain 100% coverage, all conditions must take all possible outcomes. However, to obtain non-zero coverage, one condition only needs to take one outcome.

Each condition is a boolean condition. If it evaluated to both true and false, it is marked in green. If it evaluated to either true or false, it is marked in yellow. It if did not evaluate to true or false, it is marked in pink.

To see the actual boolean value that the condition evaluated to, place your cursor above it. The value will be shown in a tool tip.



```
}

// Get acount number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
                              t*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPassword()) != 0))
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
```

Partially covered simple condition
Values tested:
{ false }

Simple Condition Coverage 50% [6/12 condition outcomes]

```
ATM -- 50% [6/12 condition outcomes]
    Bank.cxx -- 33% [2/6 condition outcomes]
        Bank::getAccount(int, std::string) -- 33% [2/6 condition outcomes]
    ATM.cxx -- 50% [2/4 condition outcomes]
    BaseDisplay.cxx -- 100% [2/2 condition outcomes]
```

Increasing Coverage

Strategies for improving coverage are discussed in Improving Test Coverage.