PLANitIO Unit Tests and Integration Tests

[1. Unit Tests and Integration Tests](#_Toc40084590)

[2 Integration Tests](#_Toc40084591)

[2.1 Overview](#_Toc40084592)

[2.2 General Structure of Integration Tests](#_Toc40084593)

[2.3 Input File Formats](#_Toc40084594)

[1.4 Standard Results Files – Naming Convention](#_Toc40084595)

[2.5 Storing Expected Results in Memory – Link Output](#_Toc40084596)

[1.5 Storing Expected Results in Memory – Path Output](#_Toc40084597)

[1.6 Storing Expected Results in Memory – Origin-Destination Output](#_Toc40084598)

[3 Test Cases](#_Toc40084599)

[3.1 Explanatory Test](#_Toc40084600)

[3.1.1 test\_explanatory](#_Toc40084601)

[3.1.2 test\_explanatory\_attempt\_to\_change\_locked\_formatter](#_Toc40084602)

# Unit Tests and Integration Tests

*Unit tests* confirm that individual methods generate the correct result from a known set of input values. They are used to confirm that methods or classes are working correctly, rather than the whole application. If code changes cause a unit test to fail, the developer can immediately review the changes to see what caused the failure.

*Integration tests* test the whole application. They run the whole application using standard input files and confirm that the contents of the output files generated by the program match the expected results.

Unit testing is particularly useful when a team of several developers is working on the same piece of software. In that case each developer must write tests for his own code before integrating it into the larger codebase of the whole project.

We have not written unit tests at the method or class level for the PlanIt application. In the case of PLANit, the development team was small (2 people) and it was sufficient to use integration tests to find issues. Test failures could still be traced back to bugs which caused them.

All tests described in the rest of this document are integration tests.

# Integration Tests

## 2.1 Overview

This section begins with a general description of how the integration tests work, which is applicable to all test cases. It then goes into more detail about the contents of the input and output files which are used in the tests.

## 2.2 General Structure of Integration Tests

Integration tests must use standard input files which define the network, demands and zoning for the test case. In the case of PLANitIO all of these are defined in one XML input file (usually named macroscopicinput.xml).

The expected results should have been generated by another method. These can include:

* hand calculation (if the test case is simple);
* running the same test case in another application. All the tests in the “Route Choice” section were run on OmniTRANS, whose outputs were recorded for comparison;
* expected error messages, where a test has deliberately invalid inputs in order to test PLANit’s input validation methods.

The expected results generated from these alternative calculation methods are stored in two ways:

* CSV and XML output files;
* Java data storage objects, usually Java Maps.

PLANitIO stores results from its runs using objects called *output formatters*. It uses two output formatters:

* PLANitOutputFormatter, which writes results to CSV and XML files;
* MemoryOutputFormatter, which stores results in Java memory.

The contents of the CSV results files generated by PLANitOutputFormatter can be compared to the standard results CSV files, and the values stored in Java memory objects are compared to those stored in MemoryOutputFormatter.

The XML results files generated by PLANitIO contain details of the output configuration used in the test, the output CSV files storing the results and the columns included in the output files. But they also contain a timestamp which gives the date and time of the PLANitIO run. This timestamp should never be the same for any two different runs. The tests must check that all the contents of the created XML file *except the timestamp* are the same as those in the standard results file, but the value of the timestamp in the generated file must always differ from that in the standard results file.

Each integration test must include:

* an XML input file to define the inputs for the test case;
* Java code to run PLANit to read the input file, run the traffic assignment and save the results;
* CSV files to store the expected results (except for tests with invalid inputs to test PLANit’s input validation);
* XML output files;
* Java code to store the expected results in Java;
* Java code to compare the results from the test run with the expected results.

Java contains its own libraries which can read CSV files (Apache Commons CSV) and check that results from a test run match the expected values (JUnit).

The class PlanItIOTestHelper.java contains common code which is used to set up and run PLANit for these tests. This class defines:-

* default configuration methods which are used in many of the tests but not all of them;
* utility methods which are used for common file actions (e.g. comparing the contents of one file with another, deleting CSV results files after the test has finished with them).

The class PlanItIOIntegrationTest contains the individual test cases described in Section 3.

## 2.3 Input File Formats

Each test case has an XML input file called macroscopicinput.xml which contains its input data in the standard XML format (using the <macroscopicdemand>, <macroscopicnetwork> and <macroscopiczoning> elements.

## Standard Results Files – Naming Convention

CSV standard results files use the naming convention:

<OutputType>\_Time Period <Time period number>\_<Number of Iterations>.csv

Where:

<OutputType> is one of “Link”, “Path” or “Origin-Destination”

<Time period number> is the appropriate time period number for the assignment execution

<Number of iterations> is whatever was defined as the maximum number of iterations in the test (for most tests this is either 1 or 500).

So typical CSV output file names include “Link\_Time Period 1\_500.csv”, “Origin-Destination\_Time Period 2\_1.csv” etc.

XML output files follow a similar naming convention but do not include the number of iterations in their title (number of iterations is included in their content anyway). So the equivalent names to the above examples would be “Link\_Time Period 1.xml” and “Origin-Destination\_Time Period 2.xml”

If a test case has more than one time period it will produce more than one set of output files. Several tests have three time periods; for these tests the generated results for all time periods are checked.

## 2.5 Storing Expected Results in Memory – Link Output

Tests of the contents of the MemoryOutputFormatter for link outputs use data transfer objects (DTOs). These objects are populated with expected result values in the Java code, and then stored in Java Maps. After the traffic assignment run has finished, the values stored in the MemoryOutputFormatter can be compared to these standard results in the code.

Expected results for link output are stored in ResultDto objects. A ResultDto object is populated by its constructor call, which has the following arguments:

* startNodeId external id of start node (used to define the link segment);
* endNodeId external id of end node (used to define the link segment);
* linkFlow flow through link (output);
* linkCost cost (travel time) of link (output);
* capacity capacity of the link (input);
* length length of the link (input);
* speed travel speed of the link (input).

ResultDto objects are stored in a Java Map whose keys are run id, time period and mode. Test cases may have more than one run, time period or mode so this allows all the results to be stored for each.

## Storing Expected Results in Memory – Path Output

Tests of the contents of the MemoryOutputFormatter for path and origin-destination cost also use a Java Map to store standard results to be compared to the outputs from the traffic assignment run. But the content of the data stored in this Map is simpler than for links, since there is only one output value to be compared to the standard results.

Path output maps use five two values as keys:-

* mode;
* time period;
* iteration number;
* origin zone external Id;
* destination zone external Id.

Expected paths are stored as a String, which consists of a series of values representing the nodes in the path surrounded by square brackets e.g. "[11,1,2,4,12]". The

The Java code gets an iterator through rows of the MemoryOuptutFormatter for the columns corresponding to the keys above for the “PATH\_STRING” output property type. This gives an output path corresponding to each origin zone and destination zone (identified by external Id). The path value in each row of the MemoryOutputFormatter is compared to the standard path from the Java Map. If the paths do not agree for a given pair of origin and destination zones, the test fails.

## Storing Expected Results in Memory – Origin-Destination Output

Tests of Origin-Destination costs stored in the MemoryOutputFormatter are done in an almost identical manner to tests of the path described in the previous section. Values are stored in a Java Map by the external Ids of origin and destination zones, and these are used to identify rows in the MemoryOutputFormatter results which are iterated through.

The only difference is that origin-destination costs are stored as doubles whereas paths are stored as strings. The code simply compares the expected cost with the calculated cost, issuing a test failure if they do not match.

It was noted in Section 2.4 that Origin-Destination output files use an iteration number one lower than those for paths and links. This applies to the MemoryOutputFormatter also. The code to run the tests for Origin-Destination must decrement the iteration counter by one to identify the correct position in the MemoryOutputFormatter.

# 3 Test Cases

## 3.1 Explanatory Test

### 3.1.1 test\_explanatory

***Purpose****:*

This test corresponds to the basic example included in the ReadMe.Md file for the PLANitIO project. This illustrates the input and output files for the simplest possible network, one link with a demand of one unit across it. This allows modellers to relate the introductory example in the documentation to a real test case.

***Description:***

A close up of a logo

Description automatically generated

The link has a demand of 10 units from Node 1 to Node 2. The road is single-lane, one-way from Node 1 to Node 2. The capacity of the link is 2000.

***Location:***

src\test\resources\testcases\explanatory\xml

All input and output files are in this directory.

*Notes:*

This test has turned out to be very useful *because* it is so trivial. If you make a code change which causes this test to fail, you know you have made a mistake or mistype in your code changes to cause the failure. And it is usually easy to trace back to the coding error which causes a wrong result in this case.

### 3.1.2 test\_explanatory\_attempt\_to\_change\_locked\_formatter

***Purpose:***

This test validates that PLANit’s formatter locking validation is working.

When PLANit runs a traffic assignment execution, it locks the setup of all its output type configurations. Any attempt to add properties to or remove properties from an output type configuration after this should throw an exception. This test includes code to add properties to the link output type configuration after the traffic assignment has been run, which should generate an exception.

***Description:***

As “test\_explanatory” above

***Location:***

src\test\resources\testcases\explanatory\xml

All input and output files are in this directory.

***Notes:***

If this test works correctly, it should run the traffic assignment once and then throw an exception. This first run will generate output files which should match “test\_explanatory” above, but it should throw the exception without writing any results to the MemoryOutputFormatter.