CANTATA

Feature Briefs

AutoTest

version 25.07

Cantata provides a full and unique suite of intelligent testing capabilities accelerating unit and integration testing of C and C++. Cantata AutoTest creates complete Cantata unit test scripts from source code using an algorithm that automatically generates Cantata test case vectors that exercise all code paths, check data, parameters and call orders. In this Feature Brief we examine the reasons why AutoTest may be used and the benefits that arise.

Document valid for versions from 22.10

Generated 03 July 2025

QA SYSTEMS
The Software Quality Company

Contents

If you have any enquiries, please do not hesitate to contact your supplier. If your product was supplied to you directly by QA Systems, the QA Systems Technical Support team can be contacted via support@qa-systems.com. Our Technical Support website page is https://qa-systems.com/support

Autotest - Feature Brief

Cantata provides a full and unique suite of intelligent testing capabilities accelerating unit and integration testing of C and C++. Cantata AutoTest creates complete Cantata unit test scripts from source code using an algorithm that automatically generates Cantata test case vectors that exercise all code paths, check data, parameters and call orders. In this Feature Brief we examine the reasons why AutoTest may be used and the benefits that arise.

Cantata provides a full and unique suite of intelligent testing capabilities for the efficient unit and integration testing of C and C++. This feature brief highlights the Cantata AutoTest capability.

Cantata AutoTest is intended to be used on C or C++ source code compiled by a C or C++ compiler.

Cantata AutoTest creates complete Cantata unit test scripts from source code using an algorithm that automatically generates Cantata test case vectors that exercise all code paths, check data, parameters and call orders. The definition of 'all code paths' used to generate the test cases is determined by selecting the metric types (Entry-point, Statement, Decision or unique cause MC/DC) in a code coverage Rule Set. The style and thoroughness of Cantata AutoTest generated test cases are also determined using standard Cantata workspace preferences.

As the AutoTest generation of Cantata tests is automatically derived from the source code, the successful generation of test cases which exercise all code paths may be limited by several technical factors in the source code which are addressed in sections 4 and 5.

In the following sections we examine the reasons why AutoTest may be used and the benefits that arise.

Reducing Reliance on System Testing

System testing is an expensive and time-consuming activity which is not designed to be an exhaustive test of all the code. Relying on system tests alone for regression testing to detect problems in previously working code is both costly and ineffective.

A suite of unit tests for all code is more effective and thorough at identifying all the affected code objects when changes are introduced, than system tests which do not execute all the code. A unit test for each object therefore acts as a valuable check that each unit continues to work after changes elsewhere which may affect them. However, if such unit tests do not already exist, writing them from scratch for any significant legacy code base is simply not commercially viable.

Cantata AutoTest automatically generates such a suite of unit tests for all selected code. Reliance on expensive and time-consuming system tests can therefore be reduced, as a safety net of regression unit tests can be run automatically and more frequently.

Supporting Continuous Integration

Continuous Integration (CI) is a technique often used in agile or iterative development methodologies where code is integrated together frequently (often leading to multiple integration builds per day). Each automated build is verified by an automated suite of unit and other tests, to detect integration errors as quickly as possible.

Cantata AutoTest generate a set of Cantata Makefiles to compile, link, deploy[1], execute and retrieve results for suites of generated Cantata tests. The Cantata Makefiles provide graphical and command line control facilities for running and reporting on a suite of Cantata unit tests. The Cantata Makefiles may be configured with the Cantata Test Script Manager and are easily integrated with any open source or commercial CI tools.

Closing Coverage Gaps in Requirements Based Tests

After requirements have been tested with Cantata, it is possible that some code constructs may remain untested. A code coverage target (using a Cantata coverage Rule Set) may be used as one measure of test completeness. Where the target is not reached, and the untested code is not redundant, Cantata AutoTest may be used to plug the coverage gaps.

Cantata AutoTest may either generate a separate test script, or added as test cases within an existing Cantata test script. This provides powerful capability to automatically generate passing test cases to fully exercise the code, as a supplement to requirements-based testing.

Identifying Testability Issues with Source Code

As a by-product of automatically generating a set of Cantata code path execution tests from source code, Cantata AutoTest will highlight various potential testability issues with the code which may prevent achieving a full safety net of passing tests. The following issues will be recorded in the Cantata AutoTest Generation Report and are detailed further in section 4.

- Dynamically Unreachable code
- Scenarios which may cause the code to crash
- Type Truncation
- Uninitialised Data
- Function Static Data
- Non-returning Functions
- Implicit Function Declarations

Where incomplete or no tests are generated by the algorithm, these have intrinsic value. Warning messages are written to the Cantata AutoTest Generation Report, showing where and why it was not possible to generate test cases, indicating possible problems in the source code or the ability to dynamically test it.

[1] Where these commands can be invoked from the command line

Changing Unit Testing Tools

Where unit testing tools are already in use (open source XUnit tools, or commercial tools) there can be several reasons for considering changing to alternative tools. Support and maintenance headaches, the tools do not provide desired productivity, capability or integration with other tools, are not qualifiable for product certification needs etc. However, investment in the test cases produced by existing tools may prevent changing to something better.

XUnit tests can be incorporated within Cantata test scripts and further enhanced by additional Cantata capabilities (allowing the retention of investment in XUnit test cases). However, migrating test cases from one proprietary commercial tool to another is not usually viable.

Cantata AutoTest provides a highly automated route for changing unit test tools. The existing tools will have been used to demonstrate the quality status of the code base, but may have limitations. Where the software is of an accepted state, Cantata AutoTest can completely replace the existing unit testing tools, by providing a suite of maintainable Cantata tests to exercise all the functionality tested by the replaced tool. Depending on the test generation preferences used, because of Cantata's unique intelligent testing capabilities, Cantata AutoTest can provide a more thorough set of tests (exercising more of the code and achieving higher levels of code coverage, and validating more global data, calls and parameters) as a regression safety net for future code changes than existing unit test tools.

Running AutoTest

Using the Cantata IDE

To use Cantata AutoTest for a set of source code files, follow the usual instructions for creating a project and generating Test Scripts and select the "AutoTest" option on the first page of Test Script Wizard. The tests will be generated according to the call interface control options specified in the wizard and the Cantata Code Generation options in Eclipse Workspace Preferences.

Alternatively, test cases can be added to an existing Test Script by choosing the "AutoTest" option when adding test cases in the Test Script Manager.

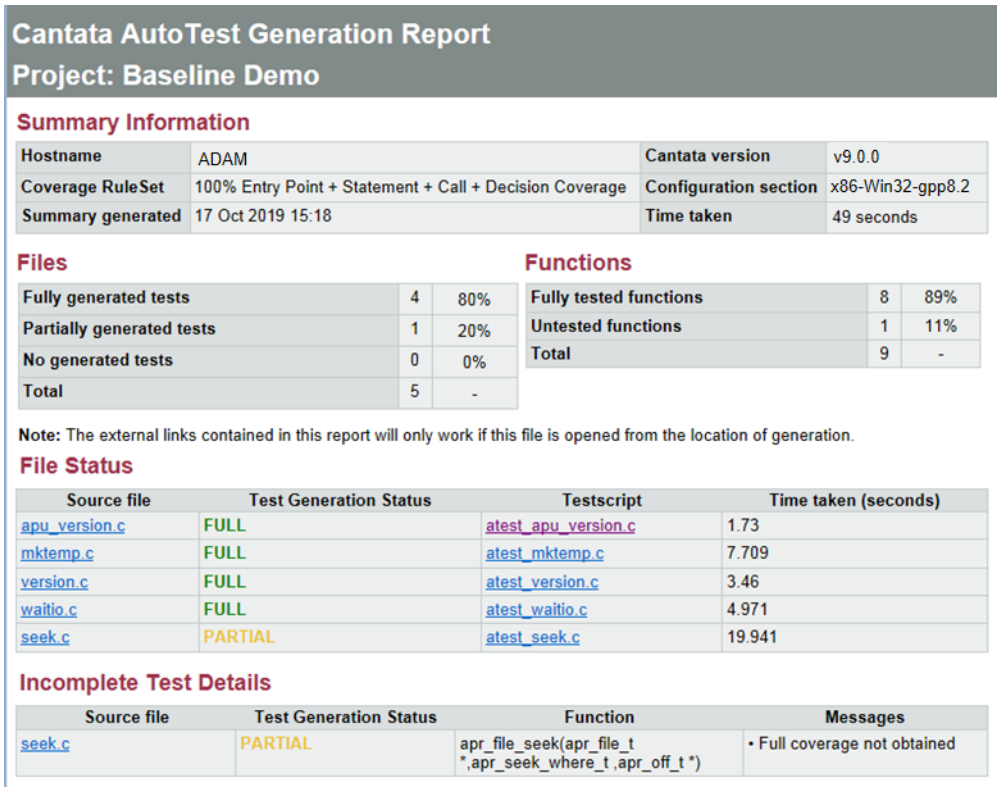
Using the Command Line

For larger code bases, it may be more convenient to run AutoTest from the command line. Invoking this with an existing build system is as simple as prefixing the compile command with the Cantata options. Code maybe selected by source directories, files and functions, with progress information displayed in the console.

Licensing

Cantata AutoTest is licensed using a separate license feature (CANTATAPP_BASELINE_TEST). If this feature is not available, the AutoTest option will be disabled in the IDE and the command line tool will report an error.

AutoTest Generation Report



Cantata AutoTest Generation Report
Project: Baseline Demo

Summary Information

Hostname	ADAM	Cantata version	v9.0.0
Coverage RuleSet	100% Entry Point + Statement + Call + Decision Coverage	Configuration section	x86-Win32-gpp8.2
Summary generated	17 Oct 2019 15:18	Time taken	49 seconds

Files

Fully generated tests	4	80%
Partially generated tests	1	20%
No generated tests	0	0%
Total	5	-

Functions

Fully tested functions	8	89%
Untested functions	1	11%
Total	9	-

Note: The external links contained in this report will only work if this file is opened from the location of generation.

File Status

Source file	Test Generation Status	Testscript	Time taken (seconds)
apu_version.c	FULL	atest_apu_version.c	1.73
mktemp.c	FULL	atest_mktemp.c	7.709
version.c	FULL	atest_version.c	3.46
waitio.c	FULL	atest_waitio.c	4.971
seek.c	PARTIAL	atest_seek.c	19.941

Incomplete Test Details

Source file	Test Generation Status	Function	Messages
seek.c	PARTIAL	apr_file_seek(apr_file_t *,apr_seek_where_t ,apr_off_t *)	• Full coverage not obtained

Summary Information

The Cantata AutoTest Generation Report contains summary information on:

- Hostname (the machine where the tests were generated) Note that links contained in the report require the machine where the tests were generated
- Cantata code coverage Rule Set used to define the required code coverage metrics and targets for a passing test
- When the report was generated
- Cantata version used
- The Configuration Section used to define the deployment of Cantata used
- The time taken to generate the tests

Summary information is also produced for the Files and Functions from which Cantata AutoTest generated tests:

- Fully Tested Files/Functions: Counts and % of files/functions where the generated tests will exercise 100% of the coverage metrics types.
- Partially Tested Files/Functions: Counts and % of files/functions where the generated tests will exercise less than 100% of the coverage metric types
- Untested Files/Functions: Counts and % of files/functions where no tests were generated

Detailed Information

The Cantata AutoTest Generation Report contains the following information for Fully Tested, Partially Tested and Untested code:

- Source file
- Function
- Messages
- Time Taken to generate the test

The content of the Warning Messages are detailed in sections 4 and 5.

AutoTest Generation Algorithm

This section describes how the Cantata AutoTest generation algorithm determines test cases. Success will result in passing Cantata unit tests, which exercise all paths through the code and undertake the style of testing specified in the test code generation preferences (see section 3.2 below).

The proportion of Cantata tests which pass will depend on the generation preferences and in particular on the threshold target for code coverage set, as well as general testability of the source code.

However, where incomplete or no tests are generated by the algorithm, these are not without value. Warning messages are written to the Cantata AutoTest Generation Report. These warnings identify where and why it was not possible to generate test cases, indicating possible problems in the source code or the ability to dynamically test it. More detail on these warning assets produced for incomplete tests is contained in section 4.

Inputs & Outputs

Inputs to the software under test consist of:

- Parameter values to function under test;
- Global data values;
- Return values from function calls;
- Output parameter values (i.e. non-constant parameters of pointer type) to function calls.

These are the inputs which Cantata has access to in order to force execution down particular paths through the source code.

The Cantata test scripts may be configured to produce checks which pass on the following outputs to verify the software under test:

- Return value from the function under test;
- Output parameter values from the function under test;
- Global data values;
- Order of function calls made;
- Parameter values to function calls.

If, during future development, the logic of the software under test changes these outputs then the tests will fail. The developer can then analyse the results to determine whether this change in outputs is expected, or whether it indicates that a bug in previously working code has been introduced.

Generation Preferences

Cantata AutoTest uses Cantata Code Generation options in Eclipse Workspace Preferences These preferences include:

- The level of code coverage required (Cantata coverage Rule Set);
- How function call interface control should be exercised (i.e. whether the calls are stubbed, wrapped, isolated or whether a real implementation should be used);
- Whether to access global static variables;
- Whether to isolation test static functions directly, or indirectly as a cluster through calling non-static functions;
- What data to check (i.e. global data, function call arguments or return values);
- Modify Global Data in function calls - whether stubs/wrappers should always modify global data to force paths;
- Time Limit (per function) - maximum calculation time;
- Path Limit - maximum number of statements to execute on each test path;
- Decision Limit - maximum number of decisions to execute on each test path;
- Maximum Array size - maximum size in bytes a map area can take.

These test code generation preferences affect the style of the Cantata tests and limit the test generation scope. Tests can be more or less thorough, unit isolation or cluster within a file tests, and can use white-box or black-box approaches. The decision on what style of unit tests to generate with Cantata AutoTest will depend on regression plan objectives and how code changes, including refactoring approaches will be implemented after the Cantata tests are generated.

Generated Test Cases

Path Solving The first default test case generated by Cantata AutoTest sets all inputs to zero. This provides the initial route through the software under test.

The algorithm then examines unexecuted paths through the software under test and attempts to find inputs that force execution down these paths. Once all areas of the software under test have been exercised or all possible paths have been tried, the algorithm ends, and the resulting test cases are generated into a test script. When all selected source code has had a test script generated, a Cantata AutoTest Generation Report is created.

Test Case Structure The AutoTest generated test cases follow the standard structure of a Cantata test case with a local variable to store the value of each parameter; an expected calls list; and the assignments and function calls associated with checking global data. If the software under test returns a value, it will be stored and checked immediately afterwards. Structures are checked field-by-field. If the returned value is not known

(i.e. the value was uninitialized in the current path) the check is omitted, and a warning recorded for the uninitialized variable and that the check is not present.

Function Calls Simulating/Intercepting Function Calls

The Cantata default Code Generation preference is that all function calls are simulated or intercepted with an isolate, stub or wrapper, allowing the order of calls and the arguments to be checked and any necessary outputs to be set.

Isolation and Cluster Testing

The default behaviour is to generate tests for each function in isolation. However, if Call Interface Control preferences are set not to generate stubs, wrappers or isolates then the actual function within the file will be called. In this case, the called function will be treated as part of the software under test and its definition will be added to the paths to solve. This is known as cluster testing.

Checking and Setting Data

Each call to a function that is stubbed, wrapped or isolated has an associated instance in the test script. By default, each instance checks input parameter values (if there are any) and sets the return value and output parameters (if there are any).

If an input parameter is a pointer, both the address and its contents are checked. Checks are added for each field/element in structure/array parameters.

The algorithm considers any non-constant pointer used as an argument in a function call to be an output parameter - an input to the software under test that can be used to force paths. Each instance in the stub/wrapper/isolate will therefore contain assignments to these memory addresses so that they can be used to control the path after the function call.

The simulated function is also able to control the execution path of the software under test by modifying global data.

C++ Elements AutoTest supports the following elements for C++ 03 and earlier C++ versions:

- C++ concrete & abstract base classes
- Overloading and inheritance
- Namespaces and classes
- Exception handling
- Templates when explicitly instantiated within the given code
- Mixed C & C++ code bases

Cantata AutoTest can be used to test C++ code containing elements/versions other than the ones listed above. Tests for supported C++ code will be auto-generated, and additional test cases can then be added.

Assets Produced for Incomplete Tests

Where Cantata AutoTest results in incomplete or no tests, warning messages for each function are written to the AutoTest Generation Report. As these warning messages indicate possible problems in the source code or its dynamic testability, they can be of significant value.

Unreachable Code

Cantata will report a failure if it cannot generate test cases to exercise all parts of the software under test, depending on the level of code coverage being used. If there are areas of "dead code" in the software under test, they will be reported as a warning and, when run, the test script will show a coverage failure.

Unreachable code may not always be obvious, and can be the result of:

- Mutually exclusive conditions;
- Complicated Boolean expressions with repeated terms;
- Pre-processor macro expansions abstracting the logic of the code;
- Very defensive programming.

Crash Scenarios

Whilst the different sets of inputs are being explored, the algorithm may discover paths that cannot be used in a test because, when evaluated, the result would be undefined. Therefore, if a set of inputs results in a path with one of the following, the path is abandoned:

- Dereferencing a null pointer;
- Array index out of bounds;
- Uninitialised variables used in decisions;
- Dividing by zero.

Details of these situations are logged as a warning as full coverage can often be obtained via other paths. However, the presence of these warnings mean that it is technically possible to cause undefined behaviour in the software under test and the user may wish to consider adding extra code to prevent it.

It is important to note that although the algorithm reports that potential problems exist, this may not be the complete set of all such possible situations.

Type Truncation

If variables are cast to a type of a smaller size, data may be lost and the algorithm cannot guarantee to correctly determine the resulting value as this will depend on the compiler/platform used. These situations are reported as warnings, as it can be indicative of an error in the code, and the algorithm will continue to solve the path.

Uninitialised Data

If a variable declared in the software under test is used before being initialised, then a warning message is logged in the AutoTest Generation Report. This problem can arise with structures, where unused fields are left un-initialised.

Function Static Data

Non-constant static variables defined inside the software under test will result in test cases not produced for functions using them, as these variables maintain their values between test cases and hence break the independence of each test case. Paths which encounter a non-constant static variable will be abandoned and a warning logged. However, if the variable is constant, it is permitted as the value will remain the same for each test case.

Non-returning Functions

If the software under test calls the exit method or a function that is marked with the noreturn attribute (i.e. the function called is not expected to return control back to the caller) the generated test script will not take this into account. Instead, the non-returning function will be simulated and control will be passed back to the software under test. A warning is raised in this case.

Control being passed back to the calling function may result in unexpected behaviour if the code was written under the assumption that execution would end. In this case conditional compilation using the ***_QAS_CANTATA_*** macro can be employed to allow the tool to generate a meaningful test.

Implicit Function Declarations

Cantata AutoTest requires that all function calls have a forward declaration of the function; it cannot rely on implicit declarations that are allowed by many compilers. If a call to a function that has not been declared, paths containing the call will be abandoned and a warning logged.

Known Limitations and Workarounds

Cantata AutoTest has some known limitations in generating full passing tests for certain code. Each of these limitations, if encountered in the source code, will result in a warning message for each function where the limitation was encountered being written to the Cantata AutoTest Generation Report.

File containing Program Entry Points

Cantata test scripts define a program entry point (main in most cases) so they can be built into standalone unit test executables. However, this means that to avoid multiple definition errors, if a source file contains the entry point then Cantata AutoTest will not generated test cases for any functions defined within it.

The workaround for this is to rename the entry point in the software under test using the *_QAS_CANTATA_* macro in conditional compilation for testing purposes.

Long Jumps

Use of the setjmp and longjmp functions in setjmp.h does not have special meaning to the Cantata AutoTest algorithm. Therefore, the function calls are simulated and the jump will not occur.

The workaround for this is to use the *_QAS_CANTATA_* define in conditional compilation for testing purposes, as when unit testing, the effect of a call to longjmp should be treated as if the function has exited.

Functions Defined in Header Files

In order for a test script to have access to the same types as the software under test the test script will include the same header files as the source file. This means that if one of these header files contains a function definition then a multiple definition error will occur when the test script and source code object files are linked.

The workaround is to modify the software under test and add a pre-processor define to change whether the header file defines or simply forward-declares the functions.

Looping

Looping constructs in the code require the algorithm to "unwrap" the code to determine suitable inputs for the test case. The more iterations around the loop, the more complex this becomes and the algorithm will take longer to solve the problem. The problem increases when loops are nested inside other loops. The algorithm has a default termination time of 20 minutes for calculating paths for each function before moving onto the next, and this can be hit where loops are very complex.

The workaround is to increase the termination timeout or to consider whether the code is too complex to be easily understood and maintained in the future

Inline Assembler

Cantata does not verify any operations carried out from inline assembler code and will report a warning if an asm declaration is found. Cantata will ignore any code within an assembler block and generate a test script as if it were not there. There is no workaround.

Standard Predefined Macros

Due to their platform-specific or unpredictable values, Cantata is unable to generate tests for functions that make use of the standard predefined macros such as *_FILE_* and __TIME__.

The workaround is to use the *_QAS_CANTATA_* define in conditional compilation to turn these predefined macros into constant known strings for testing purposes.

Memory Checking of Unions and Structures

Cantata uses a CHECK_MEMORY directive to verify the contents of global variables that are of union or structure type. The format of the expected global variable is initialised by the test script. However, if such a global variable is initialised within the software under test in certain ways, compiler padding may result in a failing check of the actual data against the expected data.

The workaround is to manually modify the expected data in the generated test script.

Compound Statement Expressions

The GNU extension shown below, allowing compound statements in an expression is not currently supported and will result in paths being abandoned by the algorithm and a warning reported.

int value = ({ int x = 0; x++; x; });

There is currently no workaround.

Nested Designated Initialisers

The use of designated initialisers for fields in composite types is supported. However, if fields/elements in nested types are of the following form, paths using these types will be abandoned and a warning logged.

struct MyStruct s = { .f1.sf1 = 1; .f1.sf2 = 2; };

The workaround is to change the code to use the following alternative form of initialisation:

struct MyStruct s = { .f1 = { .sf1 = 1; .sf2 =

