



## IBM® Rational® Rhapsody® TestConductor Add On



### Code Coverage Limitations

*Rhapsody*<sup>®</sup>

**IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup>  
TestConductor Add On**

**Code Coverage Limitations**

**Release 2.8.0**



## License Agreement

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of the copyright owner, BTC Embedded Systems AG.

The information in this publication is subject to change without notice, and BTC Embedded Systems AG assumes no responsibility for any errors which may appear herein. No warranties, either expressed or implied, are made regarding Rhapsody software including documentation and its fitness for any particular purpose.

## Trademarks

IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup>, IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup> Automatic Test Generation Add On, and IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup> TestConductor Add On are registered trademarks of IBM Corporation.

All other product or company names mentioned herein may be trademarks or registered trademarks of their respective owners.

© Copyright 2000-2017 BTC Embedded Systems AG. All rights reserved.

# Contents

---

## Content

Contents.....	4
Contacting IBM® Rational® Software Support.....	5
General information .....	6
General Limitations.....	7
Limitations regarding C.....	8
Limitations regarding C++.....	10
Tested Compilers.....	13
MC/DC-Coverage.....	15

# Contacting IBM<sup>®</sup> Rational<sup>®</sup> Software Support

---

IBM Rational Software Support provides you with technical assistance. The IBM Rational Software Support Home page for Rational products can be found at <http://www.ibm.com/software/rational/support/>.

For contact information and guidelines or reference materials that you need for support, read the [IBM Software Support Handbook](#).

For Rational software product news, events, and other information, visit the [IBM Rational Software Web site](#).

Voice support is available to all current contract holders by dialing a telephone number in your country (where available). For specific country phone numbers, go to <http://www.ibm.com/planetwide>.

Before you contact IBM Rational Software Support, gather the background information that you will need to describe your problem. When describing a problem to an IBM software support specialist, be as specific as possible and include all relevant background information so that the specialist can help you solve the problem efficiently. To save time, know the answers to these questions:

What software versions were you running when the problem occurred?

Do you have logs, traces, or messages that are related to the problem?

Can you reproduce the problem? If so, what steps do you take to reproduce it?

Is there a workaround for the problem? If so, be prepared to describe the workaround.

# General information

---

TestConductor can compute the code coverage achieved by executed test cases. The code coverage feature of TestConductor is based on source code instrumentation. Code coverage computation is restricted to C and C++. Section “Limitations regarding C” contains known limitations regarding C language. Section “Limitations regarding C++” contains known limitations regarding C++ language. Section “Tested Compilers” contains a list of tested compiler versions.

# General Limitations

---

- 1) Computation of code coverage is not supported for Rhapsody Eclipse platform integration.
- 2) Computation of code coverage is not supported when testing AUTOSAR software components.

# Limitations regarding C

---

- 1) The C code coverage feature is limited to C language constructs defined in the C90 standard (ISO/IEC 9899:1990), extended by the C99 and C11 compiler specific constructs listed below.
- 2) 'Designating Initializers' (a C99-extension, for instance implemented by the GCC compiler) are supported.
- 3) 'Compound literals' (a C99-extension, for instance implemented by the GCC compiler) are supported.
- 4) `__try...__finally-` and `__try...__except-` statements as well as `__leave-` statements (all three statement kinds are Microsoft-C extensions realizing a limited kind of exception mechanism similar to the C++ `try...catch-` statement) are supported.
- 5) Statement-expressions (a GCC extension of C, for instance `e=({if (x) x=y; x;});`) are supported. The surrounding `{' ... '}` is not counted as a statement (because it's a special kind of expression), but each statement within the curly brackets is taken into account for the statement coverage calculation.
- 6) A "Downcast" is defined to only occur when a cast happens from an integer or enumeration to another integer or enumeration (causing a value-change), but not for instance if a cast happens from for instance a floating point type to another floating point type involving a value change.
- 7) 'long long' (a type in the Microsoft and GNU C-dialect) is supported. `'__int64'` (a type in the Microsoft C-dialect) is supported.
- 8) Microsoft SAL-annotations may occur in the input-source, but they are not existent any more in the code instrumented by TestConductor. In most cases this should not be a problem, because SAL-annotations have no execution semantics, they are just meta-information. In addition, the macro `'_USE_ATTRIBUTES_FOR_SAL'` is set to '0' for each source file which is instrumented by TestConductor. This is also the default-setting for Visual Studio 2010, but this is not the default-setting for Visual Studio 2008.
- 9) For each translation unit instrumented by TestConductor, a line `'#define __declspec(X)'` is defined at the beginning of the translation unit, if the target is GNU/Cygwin. As a consequence, an instrumented translation unit may not rely on this Windows-specific GCC extension.
- 10) Header files are not instrumented, that is statements and expressions in header files are not taken into account for the coverage computation.
- 11) GNU case ranges are supported, by relating one GNU case range to one switch-case coverage item. There is a limitation in the generated coverage report regarding GNU case ranges: A covered GNU case range (for example 'A...'C') is always denoted by the lower bound of the range (in the example:'A'), even if this value was not covered and instead another value from the range.
- 12) Hexadecimal floating constants (new C language feature in C99) are not supported, if `LDBL_MANT_DIG` is not one of the values 53, 64 or 113.

- 13) The additional floating point types `__float128` and `__float80` introduced by GCC are not supported. Visible appearance of this type in the GNU C/C++ Standard library could be prevented by using a C/C++-dialect with the `'-std=...'` GCC command line option that does not contain this type (for instance, `'-std=c++11'` instead of `'-std=gnu++11'`).

# Limitations regarding C++

---

- 1) The C++ code coverage feature supports C++03 (ISO/IEC 14882:2003), C++11 (ISO/IEC 14882:2011) and C++14 (ISO/IEC 14882:2014). In the following, supported additional compiler extensions are listed, as well as constructs with restricted support.
- 2) C++ templates are not instrumented, but may exist in the source code.
- 3) Implicit constructs (that is, C++ language constructs which do not literally exist in the source code, but exist implicitly as dictated by the C++ standard) are not instrumented. Examples:

- Default-constructor
- implicit copy-constructor
- implicit assignment-operator
- implicit destructor

- 4) "Downcasts" appear only if the source- and destination-type of the cast expression are integral types (That is, Integer-, bool- and enumeration-Types). Note that it is possible that the cast-operand could be a call-expression of a class-member-conversion-operator, which converts the class-instance to an integral type. Example:

```
class C {public: operator short() const;};  
  
char c = c_inst; /* Downcast! */
```

- 5) "Division-By-Zero" is only detected for the builtin '/'-operator (applied on Integral- or floating-point types), but not for user-defined '/'-operators.

Example:

```
class D { public: bool operator/(int i) const; };  
  
int i;  
  
char c = d_inst/i; /* no Division-By-Zero instrumentation */
```

- 6) Conditions and decisions are only defined based on the predefined logical operators, and by their definition location (for instance: expression of if-statement). Note that suitable conversion operators allow also class-instances to appear as conditions/decisions, but defining a logical class-member-operation does not yield to an instrumentation as conditions or decisions. Example:

```
class C { public: operator bool() const; };  
  
class D { public: bool operator&&(const D& d); };  
  
C c_inst;  
  
D d_inst;  
  
bool b1 = c_inst && c_inst; /* <-- Two conditions 'c_inst', '&&'-result is decision */
```

```
bool b2 = d_inst && d_inst; /* <-- Contains neither a decision nor a condition */
```

- 7) A Cast to bool is defined not to be a downcast, if it is a decision or condition.
- 8) A "Downcast" is defined to only occur when a cast happens from an integer or enumeration to another integer or enumeration (causing a value-change), but not for instance if a cast happens from for instance a floating point type to another floating point type involving a value change.
- 9) Microsoft type extensions like '`__int64`' are supported.
- 10) Metaprogramming by 'type traits' (A C++-extension, introduced by ISO/IEC TR 19768, and implemented by for instance the GCC- and Microsoft-Compilers) is supported, but corresponding pseudo-functions (which are evaluated at compile-time) do not induce a decision (for instance the expression '`__is_base_of(A,B)`' - with A and B being names of classes - is not a decision).
- 11) Statement-expressions (a GCC extension of C++, for instance '`e=({if (x) x=y; x;});`') are supported. The surrounding '`{ ... }`' is not counted as a statement (because it's a special kind of expression), but each statement within the curly brackets is taken into account for the statement coverage calculation.
- 12) For each translation unit instrumented by TestConductor, a line '`#define __declspec(X)`' is defined at the beginning of the translation unit, if the target is GNU/Cygwin. As a consequence, an instrumented translation unit may not rely on this Windows-specific GCC extension.
- 13) Header files are not instrumented, that is statements and expressions in header files are not taken into account for the coverage computation.
- 14) GNU case ranges are supported, by relating one GNU case range to one switch-case coverage item. There is a limitation in the generated coverage report regarding GNU case ranges: A covered GNU case range (for example '`A...C`') is always denoted by the lower bound of the range (in the example: '`A`'), even if this value was not covered and instead another value from the range.
- 15) Lambda expressions (also known as 'lambda functions') are supported, but a lambda expression is not a function coverage goal. Nevertheless the body of a lambda expression is taken into account for the other coverage goal kinds.
- 16) Microsoft's 'for each' construct, as described in <http://msdn.microsoft.com/en-us/library/ms177202.aspx> is supported, but with analog restrictions as for the 'range-based for' construct mentioned above.
- 17) The C++11 feature 'constexpr' may be used, but the coverage functionality is limited to constructs outside of 'constexpr'-declared entities. That is, constexpr-declared Functions bodies and initializers of constexpr-declared "variables" are not taken into account for the coverage computation (corresponding code parts are not instrumented).
- 18) GCC allows to initialize an array class member by a constructor initializer as follows (this isn't valid C++11 code, even GCC issues the warning 'list-initializer for non-class type must not be parenthesized'), example:

```
class C {  
    int i[2];  
public:
```

```
C() : i({1,2}) {} /* defining the it instead as 'C() : i{1,2} {};' would be valid C++11 */  
};
```

This GCC extension is supported by TestConductor.

- 19) Concurrency by the new C++11 'thread'-class: the code coverage measurement is done globally (not thread-wise).
- 20) C++11 'static asserts': the assert expression is not part of the code coverage measurement.
- 21) C++11 'decltype': the argument expression (which is not executed by the decltype construct) does not count for the code coverage measurement.
- 22) C++11 'scoped and strongly typed enums' (also known as 'enum class'): Relational operators on such enums do not count for the code coverage instrumentation.
- 23) C++11 'noexcept' operator: the argument expression (which is not executed by the noexcept-construct) does not count for the code coverage measurement.
- 24) The additional floating point types `__float128` and `__float80` introduced by GCC are not supported. Visible appearance of this type in the GNU C/C++ Standard library could be prevented by using a C/C++-dialect with the '-std=...' GCC command line option that does not contain this type (for instance, '-std=c++11' instead of '-std=gnu++11').

# Tested Compilers

---

The following compilers were used to test the C and C++ code coverage feature of TestConductor:

- Microsoft Visual Studio 2008 on Windows 7 Service Pack 1 (32 bit)
- Microsoft Visual Studio 2008 on Windows 7 Service Pack 1 (64 bit)
- Microsoft Visual Studio 2010 on Windows 7 Service Pack 1 (32 bit)
- Microsoft Visual Studio 2010 on Windows 7 Service Pack 1 (64 bit)
- Microsoft Visual Studio 2012 on Windows 7 Service Pack 1 (32 bit)
- Microsoft Visual Studio 2012 on Windows 7 Service Pack 1 (64 bit)
- Microsoft Visual Studio 2013 on Windows 7 Service Pack 1 (32 bit)
- Microsoft Visual Studio 2013 on Windows 7 Service Pack 1 (64 bit)
- Microsoft Visual Studio 2015 on Windows 7 Service Pack 1 (32 bit)
- Microsoft Visual Studio 2015 on Windows 7 Service Pack 1 (64 bit)
- Microsoft Visual Studio 2008 on Windows 8.1 (64 bit, 32 bit target)
- Microsoft Visual Studio 2010 on Windows 8.1 (64 bit, 32 bit target)
- Microsoft Visual Studio 2012 on Windows 8.1 (64 bit, 32 bit target)
- Microsoft Visual Studio 2008 on Windows 10 (64 bit, 32 bit target)
- Microsoft Visual Studio 2010 on Windows 10 (64 bit, 32 bit target)
- Microsoft Visual Studio 2012 on Windows 10 (64 bit, 32 bit target)
- Cygwin 1.7.33 (32 bit) with GNU gcc/g++ 4.8.3 on Windows 7 Service Pack 1 (32 bit)
- Cygwin 1.7.33 (64 bit) with GNU gcc/g++ 4.8.3 on Windows 7 Service Pack 1 (64 bit)
- Cygwin 2.6.1 (32 bit) with GNU gcc/g++ 5.4.0 on Windows 10 (32 bit)
- Cygwin 2.6.1 (64 bit) with GNU gcc/g++ 5.4.0 on Windows 10 (64 bit)
- GNU gcc/g++ 4.4.7 on a Red Hat 6 Linux (64 bit)
- GNU gcc/g++ 4.8.5 on a Red Hat 7 Linux (64 bit)



# MC/DC-Coverage

---

The Modified Condition / Decision Coverage (MC/DC) is defined as follows in DO-178B:

**Condition:** A Boolean expression containing no Boolean operators.

**Decision:** A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition.

**Modified Condition/Decision Coverage:** Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

There exists no common, formal way how this definition could be appropriately mapped to the C- or C++-programming language level.

TestConductor's coverage functionality maps the MC/DC metrics to the C- and C++-programming language level as follows (first some terms and meta-functions are defined as a prerequisite, then the central terms 'Condition' and 'Decision' are defined):

- 1) Definition “**Boolean Operator**”: The following C-/C++-Operators are boolean operators: The logical negation operator ('!'), the logical AND operator ('&&'), the logical OR operator ('||'), the relational operators ('<', '>', '<=', '>='), the equality operators ('==', '!='), the bitwise exclusive OR operator ('^'), the "bitwise" exclusive OR assignment operator ('^=').
- 2) Definition “**Boolean Connective**”: A Boolean Operator which is neither a relational nor an equality-operator is called a “Boolean Connective”.
- 3) Definition “**Non-Boolean Operator**”: An expression operator which is not a “Boolean Operator” is called a “Non-Boolean Operator”.
- 4) Definition of the “**cast\_fold**”-metafunction: `cast_fold(e)` is `e`, if `e` is not a cast operator. Else, that is if `e` is a cast operator, let `op` be the operand of `e`. Then `cast_fold(e)` is (recursively) defined to be `cast_fold(op)`.
- 5) Definition of the “**effective\_operand\_of\_unary\_boolean\_operators**”-metafunction: `effective_operand_of_unary_boolean_operators(e)` is
  - equal to `e`, if `cast_fold(e)` is not the unary "Boolean Operator" '!'
  - `effective_operand_of_unary_boolean_operators(op)`, if `cast_fold(e)` is the unary "Boolean Operator" '!', and `op` is the operand of the "Boolean Operator"

- 6) Definition “**Maximal Boolean Expression**”: A “Boolean Operator” e is a “Maximal Boolean Expression”, if there exists no other “Boolean Operator” where for its operand op the following is true: `cast_fold(op)==e`

Informative note: That is, a “Maximal Boolean Expression” is a “Boolean Operator” that is not an operand of a “parent” boolean operator, ignoring casts.

Example:

```
int x,y,z;
/* In the following, '(x<y) < z' is the only maximal boolean expression: */
f(2+((x < y) < z))
```

- 7) Definition “**Condition**”: An expression E in the C/C++ programming language is a “Condition”, if all the following is true:

- E is not a constant expression

Example:

```
int i;
i = 1 && sizeof(int); /* neither '1' nor 'sizeof(int)' is a condition */
```

- E is operand of a non-unary Boolean Connective.

Informative note: If E is not operand of a non-unary “Boolean Connective”, it is not a condition: In particular, if E is operand of a unary “Boolean Operator” e\_upper, then only e\_upper *could* be a condition.

Example:

```
int i,j;
i = !j; /* '!j' is no condition, it is a decision */
i = i && !!j; /* '!j' is no condition, but '!!j' is! */
```

- effective\_operand\_of\_unary\_boolean\_operators(E) is not a (non-unary) "Boolean Connective"

Informative note: A condition must be “atomar” concerning the non-unary “Boolean Connectives”, ignoring Cast- and unary “Boolean Operators”.

Example:

```
int i,j;
int f(int);
f(i && !(!j || i)); /* '!(!j || i)' is no condition, '!j' is a condition */
```

- 8) Definition “**Decision**”: An expression E in the C/C++ programming language is a “Decision”, if at least one of the following is true:

- E is a controlling expression of an if statement.

Example:

```

int i,j;
if (i && j) j=0; /* 'i && j' is a decision */
if (1) j=1;      /* '1' is a decision      */

```

- E is a controlling expression of an iteration statement.

Example:

```

int i,j;
void f();
/* 'i && j' is a decision in the following three lines: */
while (i && j) f();
do { f(); } while (i && j);
for (;i && j;) f();

```

- E is first operand of a conditional operator.

Example:

```

int i,j,k;

/* i is a decision: */
if (i?j:k) j=1;

/* '1?j:k' and '1' are decisions: */
i=((1?j:k)?j:k);

```

- E is a “Maximal Boolean Expression”.

Example:

```

int i,j,k;
unsigned long ul1, ul2;
/* In the following statement,
 * 'i', 'ul1', 'ul2' are conditions, and
 * 'i && (int)(ul1 || ul2)' is a decision.
 */
f(i && (int)(ul1 || ul2));

```