

IBM® Rational® Rhapsody® Automatic Test Generation Add On



Limitations

Rhapsody[®]

**IBM[®] Rational[®] Rhapsody[®]
Automatic Test Generation
Add On**

Limitations

Release 3.6



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 and its fitness for any particular purpose.

Trademarks

IBM® Rational® Rhapsody®, IBM® Rational® Rhapsody® Automatic Test Generation Add On, and IBM® Rational® Rhapsody® 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 2000-2009 BTC Embedded Systems AG. All rights reserved.

Contents

Preface	6
Contacting IBM® Rational® Software Support	6
Overview.....	7
General Limitation	7
Limitations concerning Rhapsody Features	7
Ports.....	7
Blocks and Parts	7
Template Classes	7
Stable States of the SUT.....	7
RhapsodyInC, RhapsodyInAda, and RhapsodyInJava.	7
Argument Types of Provided Messages	8
Libraries.....	9
Active Objects	9
Rhapsody Properties	9
Template Classes	10
Limitations regarding new features in Rhapsody 7.0	10
Multiple Stereotypes.....	10
Components in Packages	10
Unit Unloading/Loading.....	10
“Mixed Language” models.....	10
C++ Construct Translation	11
Not Supported C++ Constructs	11
Pointer to members	11
Type declarations within functions.....	11
asm()-statements.....	11
'dynamic_cast' expression-operations	11
Bit-fields	12
Ellipsis	12
typeid()-expressions	12
Delayed destruction of temporaries, if they are bound to a static or global reference	12
Member-operator functions ' ', ' =' and ' '	12
Extern variables of class-type with an extern constructor	13
Not Correctly Translated C++ Constructs.....	13
Class temporaries created under a conditional operator (“?”, “&&”, “ ”) when they require destruction ..	13
Same named (C-)static entities in different .cpp-files	13
Casts of integral integer types, which change the value	14
Wrongly Translated Language Constructs with Warnings	14
Initialization of virtual base class-parts	14
Types ‘unsigned int’, ‘unsigned long’, ‘long long’, and ‘unsigned long long’	15
Types ‘double’ and ‘long double’	15
C++ Standard Library Restrictions	15
Other Compilation Issues	16
Exceptions	16
Union-types	16
Floating-point constants	16
Abstraction Scenarios	16

Preface

Welcome to the limitations document for IBM® Rational® Rhapsody® Automatic Test Generation Add On (Rhapsody ATG). Rhapsody ATG is a test case generation tool using standard Unified Modeling Language™ (UML™) design notations. Using ATG, you can automatically generate test suites and perform test execution for your applications developed with the Rhapsody in C++ design tool at any stage in your development cycle.

The typical UML development process (such as the Rapid Object-Oriented Process for Embedded Systems (ROPES)) is iterative, starting with an early, fairly abstract version and progressing to more and more concrete prototypes. To test a System Under Test, use ATG in your development process to do unit testing, integration testing, or regression testing.

Rhapsody ATG is complemented by Rhapsody® TestConductor. TestConductor automatically generates test monitors and test drivers from Rhapsody sequence diagrams (SDs). During automated test execution, the generated monitors determine whether the executed model satisfies the selected SDs. ATG generates test cases that can be exported to UML sequence diagrams in order to execute test cases with TestConductor.

Contacting IBM® Rational® 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.

Overview

Rhapsody ATG analyses a Rhapsody UML model and generates test cases. For test case generation, ATG parses and analyses the Rhapsody generated C++ code. This document lists the known limitations of the ATG version V3.0.

General Limitation

If a model is deeply nested in a folder hierarchy then ATG cannot generate test cases. This problem occurs due to a known Windows limitation regarding the supported length of path names (256 characters) which is not yet overcome by ATG. A workaround is to move the model to a location which is on a higher level in the folder hierarchy.

Limitations concerning Rhapsody Features

Ports

Although ATG supports Rhapsody ports, it not yet considers the provided/required interfaces specified in the port's contract for ATG test case generation. Information about interfaces must be specified with stereotypes and/or in the ATG UI.

Blocks and Parts

Models may contain blocks and parts. ATG can generate test cases, but the translation of test cases into sequence diagrams may not work correctly in some rare cases.

Template Classes

Models may contain template classes. ATG cannot be applied to models containing template classes.

Stable States of the SUT

ATG generates inputs when the SUT reaches stable states. It might happen that an application does not reach a next stable state, for instance due to a statechart that can always fire a new transition. If an application never becomes idle, then ATG will not compute meaningful test cases.

When executing generated test-cases using the Test Conductor tool, sometimes the execution may get stuck in status ACTIVE without reporting a failure or having achieved 100% coverage of the test-case. In these cases interactively performing a step and continuing the execution will force Test Conductor to resume the test-case execution.

RhapsodyInC, RhapsodyInAda, and RhapsodyInJava.

RhapsodyInC, RhapsodyInAda, and RhapsodyInJava are not supported.

Argument Types of Provided Messages

Provided messages are events, primitive operations, or triggered operations that class/component under test provides to its environment (i.e. objects and components that interact with it). For instance, if a class *C* provides an operation *foo(int a, char b, double c)*, then other components can invoke this operation on *C* with specific values for *a*, *b*, and *c*. ATG simulates such calls for test case generation with appropriate values for *a*, *b*, and *c*.

The following Rhapsody predefined types for message arguments are supported.

bool	RhpInteger
char	RhpPositive
char*	RhpReal
double	RhpString
int	RhpUnlimitedNatural
long	short
long double	unsigned char
OMBoolean	unsigned int
OMString	unsigned long
RhpBoolean	unsigned short
RhpCharacter	

The following types are not supported:

RhpAddress
RhpVoid
Void
Void *

Also supported are typedefs that use these predefined types and enumeration types. Not supported are for instance arguments with pointer types, e.g. *op(user_type *a)*. ATG cannot yet generate values for pointer arguments of provided messages.

This limitation also applies for user-defined enumeration types. Qua default, Rhapsody represents operation arguments of user-defined enumeration types by references as arguments. Such references have to be treated as pointers by ATG, unless property `CPP_CG::Type::In` redefines the representation of such arguments. In general, if a Rhapsody type (model-view) turns into a reference- or pointer type at code-view, this type is unsupported to be used as message argument for a provided message. Additionally, ATG does in general not support structured types (class/struct/unions) to be used as argument-types of interface-functions (even, if structured objects are directly passed to the function, without an indirection by pointers or references).

Another limitation regarding types concerns empty-strings of type `OMString`, `RhpString`, and `char*`. Rhapsody's Sequence Diagrams are currently not capable of representing empty strings, i.e., even though ATG generates test-cases regarding empty-strings correctly, these test-cases cannot be executed successfully using TestConductor unless empty strings are replaced by non-empty strings.

Variables and constants of integer types (cf. section 2.9) are translated wrongly into ATG, if their value is outside of the range of signed 32-bit integer. In addition, ATG could only

operate on signed 32-bit integer values. Therefore the compiler could be configured such that a warning-message is printed, if expressions exists which have these kind of types.

Please note that these limitations apply only to messages that are provided to other components by the class/component under test. It is not an issue if pointer arguments are used inside a class/component under test.

For the ATG interface definition, a value of type 'wchar_t' or 'char' can only be specified by an integer constant, but not by a character constant of the form "L'<character>'" or "'<character>'". A value of type 'float' can only be specified by a floating constant of type double, but not by a floating constant of type float (that is, the suffix 'f' or 'F' must be omitted).

Libraries

As mentioned above, ATG parses and analyses the Rhapsody generated C++ code in order to generate test cases. If the complete behavior is specified in the model, then ATG generates test cases. Note that Rhapsody users can also add libraries to the application when the application is linked in order to create an executable. Source code of linked libraries is usually not available. On the other hand, ATG requires the full source code for the analysis. Otherwise ATG cannot analyze the full behavior. This means, if some behavior is linked to the application under test with compiled libraries, then ATG cannot yet analyze the code and cannot generate test cases.

ATG will generate stub source code in order to provide default behavior for functions that are implemented in libraries. The stub code is used for actual test case generation. Users can modify and extend the stub code.

Please note that test case execution can clearly be done on the production code where libraries are linked to the class/component under test.

Active Objects

Rhapsody ATG can handle programs that use up to 10 active objects, i.e., classes with concurrency 'active'. If more then 10 active objects are alive at once, the achieved coverage may be low.

Rhapsody Properties

Rhapsody ATG uses own versions of the Rhapsody framework classes and functions. Additionally, the ATG Rhapsody framework does not contain all classes and functions of the original Rhapsody framework. Due to this approach, several values of project properties that influence the code that is generated for a Rhapsody model are not supported by ATG. The following list contains some restrictions concerning values of several Rhapsody properties:

- CPP_CG::Microsoft::EntryPoint must be set to "main" (the default-value)
- CG::Configuration::CodeGeneratorTools must be set to "internal" (default)
- CPP_CG::Configuration::ContainerSet must be set to "OMContainer" (default)

- `CPP_CG::Microsoft::IsFileNameShort` must be unchecked (default)
- `CPP_CG::Microsoft::ImpExtension` must be set to “.cpp” (default)
- `CPP_CG::Microsoft::CompileSwitches` may not be changed by the user
- `CPP_CG::Class::AdditionalBaseClasses` may not be used (default)
- `CPP_CG::Configuration::DefaultSpecificationDirectory` may not be changed (default)
- `CPP_CG::Configuration::DefaultImplementationDirectory` may not be changed (default)
- `CPP_CG::Configuration::EmptyArgumentListName` may not be changed (default)
- `CPP_CG::Microsoft::SpecExtension` may not be changed (default)
- `CPP_CG::Statechart::StatechartStateOperations` may not be changed (default)
- `CPP_CG::Operation::ImplementationName` may not be changed (default)
- `CPP_CG::Class::SpecIncludes` may only refer to files in the default spec directory
- `CPP_CG::Class::ImpIncludes` may only refer to files in the default implementation directory

Template Classes

Rhapsody Template Classes (that is, Rhapsody Classes of class type ‘Template’) are not supported.

Limitations regarding new features in Rhapsody 7.0

Multiple Stereotypes

Only the first stereotype of an element is considered by ATG

Components in Packages

Components which reside somewhere inside of packages are not visible for ATG. This means that ATG is not applicable for configurations within such components.

Unit Unloading/Loading

The results generated by ATG represent results corresponding to the state of the model at the time of test case generation. Unloaded model elements are treated the same way as deleted model elements.

“Mixed Language” models

ATG can only generate test cases for configurations within C++ components opened in RhapsodyC++. Even this is currently not working since there is a bug in the COM API function `IRPConfiguration::needsCodeGeneration` (Quintus 93745)

C++ Construct Translation

Some C++ language constructs are not yet fully supported by ATG. In the sequel we list the known limitations and give some examples.

Not Supported C++ Constructs

For a subset of C++ language constructs ATG can not yet perform an appropriate translation for the sake of test case generation. If possible ATG applies an automatic abstraction (see section *Abstraction Scenarios*) and simplification such that the compilation continues. This section lists the known not supported C++ constructs, provides some examples, and explains what kind of error a user will see.

Pointer to members

Example

```
class C {public: int i;};
int C::* ip = &C::i;
```

Error-message

No abnormal termination, instead various abstraction warnings possible.

Type declarations within functions

Example

```
int main() {
    class C {public: int i;}; /* an explicit type-declaration */
    C c;
    enum E {e1, e2} e;      /* a type-declaration embedded in another declaration */
    return e+c.i;
}
```

Error-message

No abnormal termination if the type-declaration is not done in the main()-function (instead an abstraction-warning). Otherwise (a type declaration within the main()-function): Abnormal termination with error message "Error 224: Unsupported C++ construct (a suitable abstraction can not be applied)"

asm()-statements

Example

```
void f() {asm(";");}
```

Error-message

No abnormal termination, instead an abstraction-warning.

'dynamic_cast' expression-operations

Example

```
class C {public: virtual int f();};
```

IBM® Rational® Rhapsody® Automatic Test Generation Add On (ATG)


```
class D : public C {public: int j;};
D* f () { C c; return dynamic_cast<D*>(&c);}
```

Error-message

No abnormal termination, instead an abstraction-warning.

Bit-fields**Example**

```
struct s { int i:2;} s1;
void f() { s1.j = s1.j+1; }
```

Error-message

No abnormal termination, instead an abstraction-warning.

Ellipsis**Example**

```
void f(int a ...) {
va_list ap; va_start(ap, a); char* c = va_arg(ap, char*); va_end(ap);
}
```

Error-message

No abnormal termination, instead an abstraction-warning.

typeid()-expressions**Example**

```
#include <typeinfo>
void f() {int i, j; if (typeid(i)==typeid(j)) i=0;}
```

Error-message

No abnormal termination, instead an abstraction-warning.

Delayed destruction of temporaries, if they are bound to a static or global reference**Example**

```
class C {public: int i; ~C() {i=0;}};
C f();
int j() {static C& cr=f();} /* The temporary returned by f has to be destructed delayed */
```

Error-message

"Error 14: Unsupported C++ construct (a suitable abstraction can not be applied)"

Member-operator functions '|', '|=' and '||'**Examples**

```
class C {public: int j; void operator|(int i) {j=i;}; };
int f() {C c; c.operator|(1);};
```


Error-message

Abstraction warning "ABSTRACTION 229: [...] Unsupported operator in [...]" is reported.

Extern variables of class-type with an extern constructor**Examples**

```
class C {
public:
    C() {} ;
    C(const C&); /* copy-ctor declared, not defined - that is: extern */
};
```

```
extern C c_var; /* extern variable of type C */
```

Error-message

An ATG-compile error is reported.

Not Correctly Translated C++ Constructs

For a subset of C++ language constructs ATG can not yet perform a correct translation for the sake of test case generation. This section lists the known not correctly translated C++ constructs, provides some examples, and explains the wrong semantics.

Class temporaries created under a conditional operator ("?", "&&", "||") when they require destruction**Example**

```
class C {public: int i; ~C() {i=0;}; };
C f();
void g(int i) {
    int j;
    j = (i ? f().i : 1); /* the creation of a temporary variable for the result of f() depends on 'i' */
}
```

Wrong semantics

If a temporary variable, which was created under a conditional operator, would be destructed, a warning message is printed. The temporary variable is not destructed.

Error-message

"No abnormal termination, instead an abstraction-warning"

Same named (C-)static entities in different .cpp-files**Example**

```
// a.cpp:
static int i = 1;
// b.cpp:
static int i = 2;
```


Wrong semantics

The compiler will print an error-message due to multiple definitions of 'i'.

Casts of integral integer types, which change the value**Examples**

```
short s = -1;
unsigned int ui = (unsigned int)s;
unsigned int ui2=INT_MAX;
unsigned char uc=(unsigned char)ui2; /* value-change, if: UCHAR_MAX < INT_MAX */
```

Wrong semantics

All integer-variables are internally represented as a 32-bit value. Different integer-sizes are not taken into account and integer-casts do not change the value in ATG.

Wrongly Translated Language Constructs with Warnings

For a subset of C++ language constructs ATG can not yet perform a correct translation for the sake of test case generation, and generates a warning. This section lists the known not correctly translated C++ constructs, the warnings, provides some examples, and explains the wrong semantics.

Initialization of virtual base class-parts

Virtual base classes are not fully supported in case of multiple inheritance (in 'diamond'-form), if the virtual base class has defined constructors or destructors:

Example

```
class A { public: A() {}; }
class B : public virtual A { public: B() {}; }
class C : public virtual A { public: C() {}; }
class D : public A, public B {public: D() {}; }
```

Wrong semantics

The constructor of A is not called from the constructor of D, instead it is called from B and C (that is, it is called twice). The same wrong semantics exists for destructors.

Warning message

A warning message “Object to be initialized is a virtual base class, not correctly implemented yet (the con-/destructor of a virtual base class is possibly called more than once)” is generated. This message is also printed for initializations of virtual base classes, if no 'diamond'-form exists (that is, if it does not result in wrong behavior).

Types 'unsigned int', 'unsigned long', 'long long', and 'unsigned long long'

Constants of these types are translated wrongly into ATG, if the value is outside of the range of signed 32-bit integer. In addition, ATG could only operate on signed 32-bit integer values.

Example

```
unsigned long i = ULONG_MAX; /* 0xffffffffUL */
```

Wrong semantics

The representation is truncated to 32-bit (if the previous representation has more bits) and after that the representation is interpreted as 32-bit signed integer.

Warning message

A warning message “Possible truncation from <type> to long” is generated, where <type> is 'unsigned int', 'unsigned long', 'long long' or 'unsigned long long' respectively. This warning-message is printed, if constants of these types occur.

Types 'double' and 'long double'

Constants of these types are translated wrong into ATG, if the corresponding value can not be represented in the float-range.

Example

```
double d = DBL_MAX;
```

Wrong semantics

The value is changed corresponding to a cast-operation from 'double' respectively 'long double' to 'float' (the semantics of the cast corresponds to the semantics defined by the Microsoft Visual Studio .NET 2003-compiler).

Warning message

A warning message “Possible truncation from <type> to float” is generated, where <type> is 'double' or 'long double' respectively.

C++ Standard Library Restrictions

Due to the reported restrictions (v. Wrongly Translated Language Constructs with Warnings, Not Correctly Translated C++ Constructs, Not Supported C++ Constructs) several elements provided by the C++ Standard Library (including the Standard C Library) may not work correctly. A detailed list of the support of C++ Standard Library functions can be found in two separated documents

- CppStdLibSupport_VC60.csv
- CppStdLibSupport_NET2003.csv

Each of these documents contains a table with all functions defined in the official ISO C++ Library standard. Each line of the table contains several rows that show if and how the function described in that line is supported by ATG for the two compile environments

- Visual Studio 6.0
- Visual C++ .NET 2003

In ATG we distinguish between 3 kinds of support of a function:

- Supported, meaning that one can use the function in ATG and ATG supports the same semantics for this function as defined in the standard
- Supported by Abstraction (see section *Abstraction Scenarios*), meaning that one can use the function in ATG but ATG may have a different semantics for this function
- Unsupported, meaning that one cannot use this function in ATG

To easily access the information in the table, you can open the files with spreadsheet programs like Excel.

Other Compilation Issues

Exceptions

Currently there is no full compilation support for exceptions. A workaround for this restriction was implemented, which allows exception-code at compile-time. If an exception occurs at run-time, ATG will not execute the exception handler and will continue with the test case generation.

Union-types

The 'overlapping'-semantics of unions is currently not supported by ATG. A workaround for this restriction was implemented to translate unions with struct-semantics.

Floating-point constants

The representation of floating-point constants is in general not correct (Floating-point constants are represented without the use of the exponent-'E'-syntax, in particular the representation is corresponding to the `print()`-formatstring `'%-.10f'`. That means for example that floating-point constants lose the part of the mantissa, which is (as an absolute value) less than $1E-10$).

Abstraction Scenarios

This section describes the scenarios in which ATG applies automatic abstractions in order to generate test cases.

Scenario: a variable is initialized with an expression and the expression contains a not supported C++ construct.

Abstraction: the variable will not be initialized.

Scenario: an expression statement contains a not supported C++ construct.

Abstraction: the expression statement will be ignored.

Scenario: an if-/while-/do-while-/for-expression contains a not supported C++ construct.

Abstraction: the expression will be non-deterministically chosen to be either true or false.

Scenario: the initialization of a constructor contains a not supported C++ construct.

Abstraction: the respective class element will not be initialized or all elements contained in the initialization of a constructor will not be initialized.

Scenario: the call of a constructor for a temporary variable contains a not supported C++ construct, and the temporary variable defines the value of a return-statement.

Abstraction: the constructor will not be called.

Scenario: an expression which defines the value of a return-statement contains a not supported C++ construct.

Abstraction: a NULL value will be returned.

Scenario: an expression which initializes a for-loop contains a not supported C++ construct.

Abstraction: the relevant statement will be ignored.

Scenario: an increment-statement of a for-loop contains a not supported C++ construct.

Abstraction: the relevant statement will be ignored.

Scenario: an switch-expression contains a not supported C++ construct.

Abstraction: the expression will be non-deterministically chosen from the interval [min, max]. 'min' is the smallest value of the switch-case-constants. 'max' is the largest value of the switch-case-constants.

Scenario: a switch-case constant depends on a not supported C++ construct.

Abstraction: the relevant switch-case block will be ignored.

Scenario: a call to a function with a variable number of parameters.

Abstraction: the call will be ignored.

Scenario: a call to a function which contains a type declaration in its body.

Abstraction: the call will be ignored.