IBM® Rational® Rhapsody® TestConductor Add On

Testing on a small target

Rhapsody®

IBM® Rational® Rhapsody® TestConductor Add On

Testing on a small target

Release 2.8.0

Contents

Content

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.

Introduction

This document describes how testing can be performed with IBM® Rational® Rhapsody® TestConductor Add On on a small target, while Rhapsody is running on a Windows or Linux host. We assume the basic installation is already done: The tools needed to develop software for a small target are installed (for example a target proxy server, IDE, compiler, debugger, connection to the target, etc.). Rhapsody is installed on the Windows or Linux host and a TCP/IP connection between the host and the target proxy server is available.
As small targets do not fulfill all five essential conditions for executing tests with TestConductor (file system, enough memory, execution of applications from the command line, application upload, results download) and as in addition some small targets can only be accessed via special interfaces (like JTAG) and/or via special debuggers, a target proxy server is needed as mediator between Rhapsody and the tested application on the target. So Rhapsody running on the host will not invoke the tested application on the target directly and the tested application does not itself write the test status into the file system. Instead TestConductor will use a TCP/IP connection to a target proxy which is controlling the target IDE and is responsible for collecting the information which is necessary to process the test result after test execution. The target proxy abstracts from the individual API and characteristics of the target IDE and the target being used.

Rhapsody and the target proxy can be executed on the same machine or on different machines. If running on different machines the test results can be created automatically if the target proxy can write files to the file system of the machine Rhapsody is running on.

This document describes also the protocol for the communication between TestConductor and a target proxy. A target proxy itself is not provided by TestConductor, the user has to implement a target proxy based on the specific API of the IDE which is used for development of the application. An Eclipse plugin is part of the TestConductor installation which can be used as demonstrator how to implement a target proxy.

Note that the testing on a small target is only supported for the assertion based testing mode of TestConductor. The animation based testing mode is not supported.

Communication protocol between TestConductor and the Target Proxy

As described in the Introduction above, the target proxy is a mediator between Rhapsody and the target IDE which controls the tested application on the target. Rhapsody running on the host will not invoke the tested application on the target directly. Instead Rhapsody will use a TCP/IP connection to send commands (like build, execute) to the target proxy which is responsible for actually performing these commands and for sending status information back to Rhapsody using the same TCP/IP connection. The TCP/IP commands for this communication protocol are documented in this chapter.

The target proxy is also responsible for creating files containing the data (status of evaluated assertions and code coverage information) collected during execution of the tests. The way how this data can be retrieved from the tested application depends on the capabilities of the target: If the target itself supports a file system the tested application can write into files the target proxy only needs to store the files in the file system of the host Rhapsody is running on. If the tested application cannot write files the target proxy must retrieve this data from the tested application – for example using functionality of the target IDE or a debugger attached to the tested application – and write it into files on the host Rhapsody is running on. Code which must be compiled into the tested application to be able to perform certain functionality – like sending status information of an assertion to the target proxy or exiting the application – can be defined in tags of the code generation configuration.

Figure 1 shows an overview of the communication protocol between TestConductor and the target proxy. When implementing a target proxy for a specific target, the user needs to make sure the target proxy supports the TCP/IP commands listed below and follows the specified protocol. A reference implementation of a target proxy is part of the TestConductor installation, including the Java source code. This reference implementation is described in chapter Eclipse Target Proxy.

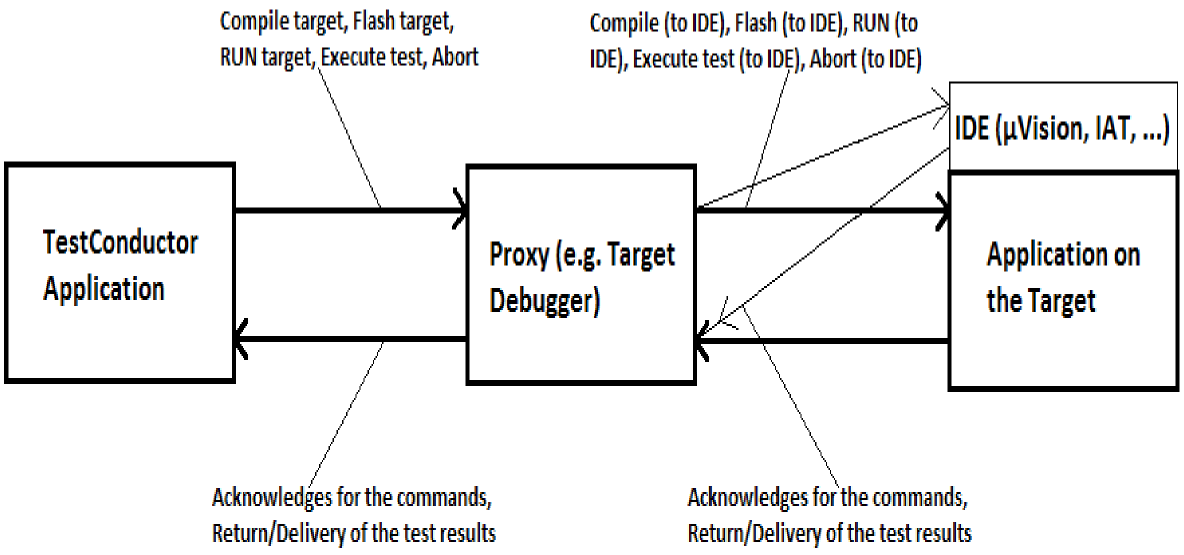

Compile target, Flash target, RUN target, Execute test, Abort
Compile (to IDE), Flash (to IDE), RUN (to IDE), Execute test (to IDE), Abort (to IDE)
IDE (µVision, IAT, ...)
TestConductor Application
Proxy (e.g. Target Debugger)
Application on the Target
Acknowledges for the commands, Return/Delivery of the test results
Acknowledges for the commands, Return/Delivery of the test results

Figure 1: Communication between TestConductor, the target proxy and the tested application on the target

TCP/IP commands sent from TestConductor to the target proxy:

1.) "**Open IDE project** (path)" ("OPEN_IDE_PROJECT|PathOfIDEProject\0")

This TCP/IP string is provided by the small targets API in order to close or to open an IDE project. The real string that is send from TestConductor to a target proxy server is "OPEN_IDE_PROJECT|PathOfIDEProject\0".

If "PathOfIDEProject" contains an empty string the target proxy should close an IDE project. If "PathOfIDEProject" is not empty the target proxy should open the corresponding IDE project.

When performing "Build TestCase|TestContext|TestPackage", TestConductor automatically sends the command to close an IDE project before code generation and sends the command to open the IDE project after code generation.
This can be switched on or off via the TargetProxyAutoOpenIDEProject tag.

As described in the "Configuration" section below the parameter "PathOfIDEProject" is specified by the TargetProxyIDEProject tag.

2.) "**Compile target**" ("COMPILE_TARGET\0")

This TCP/IP string is provided by the small targets API in order to compile the tested application. The real string that is send from TestConductor to a target proxy server is "COMPILE_TARGET\0".

When performing "Build TestCase|TestContext|TestPackage", TestConductor automatically sends this command after code generation.
This can be switched on or off via the TargetProxyAutoCompile tag.

3.) “**Flash target**” (“FLASH_TARGET\0”)

This TCP/IP string is provided by the small targets API in order to flash target (load the application on the target). The real string that is send from TestConductor to a target proxy server is “FLASH_TARGET\0”.

When performing “Build TestCase|TestContext|TestPackage”, TestConductor automatically sends this command after building the application has finished. This can be switched on or off via the TargetProxyAutoFlashTarget tag.

4.) “**RUN target**” (“RUN_TARGET\0”)

This TCP/IP string is provided by the small targets API in order to run (start) the tested application on the target. The real string that is send from TestConductor to a target proxy server is “RUN_TARGET\0”.

When performing “Execute TestCase|TestContext|TestPackage”, TestConductor automatically sends this command to the target proxy. This can be switched on or off via the TargetProxyAutoRunTarget tag.

5.) “**Execute test** (test case name, result file)” (“EXECUTE_TEST|testCaseName|resultFile\0”)

Execute test (test case name, result file, coverage file)” (“EXECUTE_TEST|testCaseName|resultFile|coverageFile\0”)

This TCP/IP string is provided by the small targets API in order to start the execution of a test case or a test context on the target. The real string that is send from TestConductor to a target proxy server is “EXECUTE_TEST|testCaseName|resultFile\0” or “EXECUTE_TEST|testCaseName|resultFile|coverageFile\0”.

The first parameter “testCaseName” contains a string for the corresponding name of the test case (if a test case or test context is executed with a restart of the executable application after each test case) or the string “all” (if all test cases of a test context shall be executed on the target without a restart of the executable application).
The second parameter “resultFile” contains the full path name of the file the target proxy should write the data about test assertions to. After the test execution has finished TestConductor will create the test result from the data in this file. The full path name of this file can be configured using the tag rtc_result_filename.
The third parameter “coverageFile” is optional. It contains the full path name of the file the target proxy should write the code coverage data to. After the test execution has finished TestConductor will create the code coverage result from the data in this file.

When performing “Execute TestCase|TestContext|TestPackage”, TestConductor automatically sends this command to the target proxy after the tested application has been started. This can not be switched on or off via a special tag.

6.) “**Abort**” (“ABORT\0”)

This TCP/IP string is provided by the small targets API in order to abort the build of the application or the execution of a test case or a test context on the target. The real string that is send from TestConductor to a target proxy server is “ABORT\0”.

TCP/IP acknowledges sent from target proxy to TestConductor:

1.) “**OK**\0” acknowledge from the target proxy

If a TCP/IP command is sent and executed successfully by the target proxy server or the target, then the string “OK\0” should be returned by the target proxy server.

2.) “**ERROR**\0” acknowledge from the target proxy

If a TCP/IP command is sent but its execution by the target proxy server or the target fails, then the string “ERROR\0” should be returned by the target proxy server. A corresponding error message is displayed by TestConductor in the Rhapsody output window.

3.) “**NOT_SUPPORTED**\0” acknowledge from the target proxy

If a TCP/IP command is sent which is not supported by the target proxy server or the target, then the string “NOT_SUPPORTED\0” should be returned by the target proxy server. A corresponding error message is displayed by TestConductor in the Rhapsody output window.

Command sequences

1.) **Build Test Case, Build Test Context, Build Test Package**

The following sequence takes place if the user performs the Build TestCase or Build TestContext or Build TestPackage command:

1.1) TCP/IP command “OPEN_IDE_PROJECT|””\0” is sent to the target proxy (if TargetProxyAutoOpenIDEProject tag is set to true), in order to close an IDE project.

1.2) Code generation. (Rhapsody API).

1.3) TCP/IP command “OPEN_IDE_PROJECT|”path to IDE”\0” is sent to the target proxy server (if TargetProxyAutoOpenIDEProject tag is set to true), in order to open an IDE project.

1.4) TCP/IP command “COMPILE_TARGET\0” is sent to the target proxy (if TargetProxyAutoCompile tag is set to true), in order to compile the TestConductor application for the target.

1.5) TCP/IP command “FLASH_TARGET\0” is sent to the target proxy (if TargetProxyAutoFlashTarget tag is set to true), in order to load the tested application to the target.

Note that, if one of this “TargetProxyAuto...” tags is unset, the user must execute the corresponding command manually.

Note also, that the complete build process ends if one of these TCP/IP commands fails (error, timeout, …). In that case an appropriate error message is displayed by TestConductor in the Rhapsody output window.

2.) **Execute Test Case, Execute Test Context, Execute Test Package**

The following sequence takes place if the user performs the Execute TestCase or Execute TestContext or Execute TestPackage command:

2.1) TCP/IP command “RUN_TARGET\0” is sent to the target proxy (if TargetProxyAutoRunTarget tag is set to true), in order to start the TestConductor application on the target.

Note that, if the TargetProxyAutoRunTarget tag is unset, the user must execute the corresponding command manually.

2.2) TCP/IP command “EXECUTE_TEST|testCaseName|resultFile\0” (or “EXECUTE_TEST|testCaseName|resultFile|coverageFile\0”) is sent to the target proxy, in order to execute a test case or a test context or test package on the target.

2.3) Result verification and generation of a HTML result report.

Note also, the complete execute process ends if one of these TCP/IP commands fails (error, timeout, …). In that case an appropriate error message is displayed by TestConductor in the Rhapsody output window.

3.) **Abort**

The following sequence takes place, if the user performs the “abort” user command:

3.1) TCP/IP command “ABORT\0” is sent to the target proxy, in order to abort the test execution.

Prototype for a target proxy

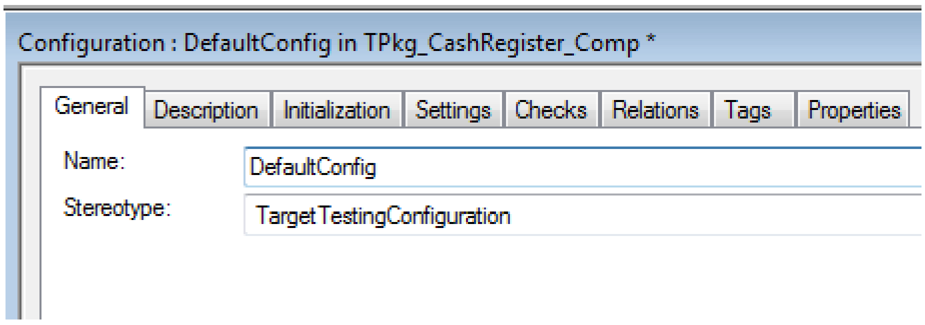
Note that a reference implementation of a target proxy is part of the TestConductor installation. It is described in the chapter Eclipse Target Proxy below.

Execution of TestCases on a Small Target

Please follow the steps as described in the sections below.

Preparing the Code Generation Configuration

- Features dialog of the Code Generation Configuration, tab General: Select the correct predefined stereotype, <<TargetTestingConfiguration>>. Deselect stereotype <<TestingConfiguration>> if it is selected.
  When using the Eclipse Target Proxy demonstrator, the stereotype <<ETPTargetTestingConfiguration>> can be used instead. This stereotype inherits from <<TargetTestingConfiguration>> and contains some predefined tag values appropriate for the Eclipse Target Proxy. For example, for functionality like exiting the tested application or storing assert information the default implementation is used.
  Note: To be able to use these default implementations the macro "TC_SMALL_TARGETS_ALLOW_DEFAULT_IMPL" must be defined when compiling the application. Add this macro to the compile switches of the code generation configuration if you want to use the default implementation for TargetProxyAssertImplementation, TargetProxyCodeCovInfoImplementation or TargetProxyExitImplementation (see documentation of these tags below).


Configuration : DefaultConfig in TPkg_CashRegister_Comp *

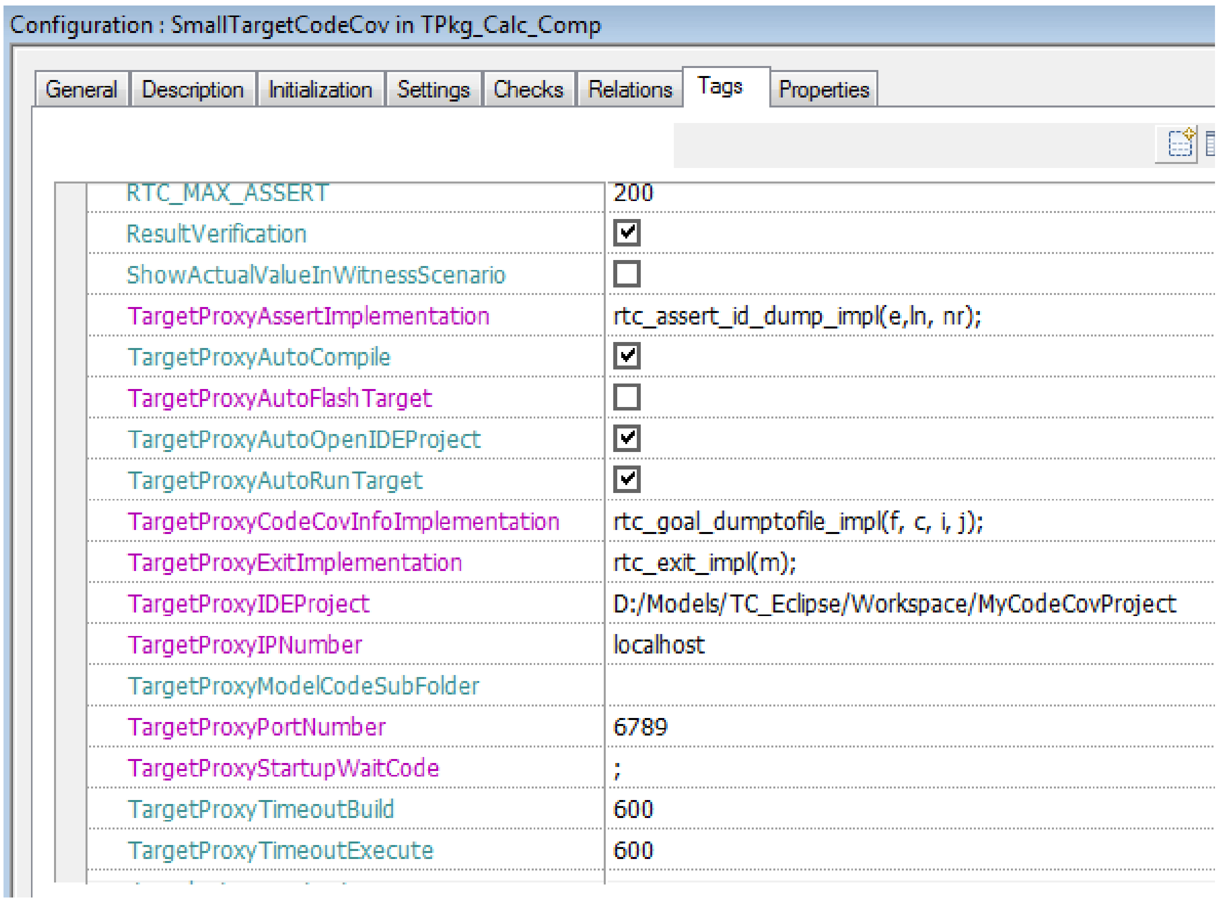
Name: DefaultConfig

Stereotype: TargetTestingConfiguration

Figure 2: TargetTestingConfiguration Stereotype

Figure 2 shows that for Small Targets the stereotype <<TargetTestingConfiguration>> must be selected.

Figure 3 shows the Settings / Tags of the stereotype <<TargetTestingConfiguration>> which are described below. In the example the values for the Eclipse Target Proxy demonstrator are shown. For other implementations of a target proxy different values might be necessary.


Configuration : SmallTargetCodeCov in TPkg_Calc_Comp

RTC_MAX_ASSERT	200
ResultVerification	☑
ShowActualValueInWitnessScenario	☐
TargetProxyAssertImplementation	rtc_assert_id_dump_impl(e,ln, nr);
TargetProxyAutoCompile	☑
TargetProxyAutoFlashTarget	☐
TargetProxyAutoOpenIDEProject	☑
TargetProxyAutoRunTarget	☑
TargetProxyCodeCovInfoImplementation	rtc_goal_dumptofile_impl(f, c, i, j);
TargetProxyExitImplementation	rtc_exit_impl(m);
TargetProxyIDEProject	D:/Models/TC_Eclipse/Workspace/MyCodeCovProject
TargetProxyIPNumber	localhost
TargetProxyModelCodeSubFolder	
TargetProxyPortNumber	6789
TargetProxyStartupWaitCode	;
TargetProxyTimeoutBuild	600
TargetProxyTimeoutExecute	600

Figure 3: TargetTestingConfiguration Settings / Tags

- Tags of the Code Generation Configuration:
  - TargetProxyAssertImplementation
    This tag can be used to specify the code which will be used to store the outcome of an assert during execution of a test case. This code will be executed every time an assert is evaluated.
    The concrete way how to collect and store the information about the outcome of an assert depends on the target, the used IDE and the target proxy and their capabilities.
  - TargetProxyAutoCompile
    Check this tag if TestConductor should send the command to build the application to the target proxy when performing “Build TestCase|TestContext|TestPackage”. If this tag is not checked the user must manually build the application in the target IDE.
  - TargetProxyAutoFlashTarget
    Check this tag if TestConductor should send the command to load the application binary on the target to the target proxy after building it. If this tag is not checked the user must manually load the application on the target.
  - TargetProxyAutoOpenIDEProject
    Check this tag if TestConductor should send the command to open the IDE project in the target IDE to the target proxy before building the application. If the tag is checked, TestConductor also sends a command to close the project

before code generation to avoid pop up windows by the IDE because of modifications of the source code files.

- TargetProxyAutoRunTarget
  Check this tag if TestConductor should send the command to execute the application to the target proxy when starting the execution of a test case. If the tag is not checked the user must start the application on the target manually.
- TargetProxyCodeCovInfoImplementation
  This tag can be used to specify the code which will be used to store collected collected code coverage information during execution of a test case.
  The concrete way how to collect and store this information depends on the target, the used IDE and the target proxy and their capabilities.
  This tag is relevant only when computing code coverage information.
- TargetProxyExitImplementation
  This tag can be used to specify the code which will be used to terminate the tested application at the end of test case execution.
  The concrete way how to to terminate the application depends on the target, the used IDE and the target proxy and their capabilities. For some targets it is not possible for an application to terminate itself. In this case, some kind of reset mechanism can be implemented and entered into this tag.
- TargetProxyIDEProject
  This tag can be used to provide the full path name of the IDE project of the application if the option TargetProxyAutoOpenIDEProject is checked. This path is also used to access the source code of the application if computation of code coverage is enabled (if the code is not compiled in the default code generation folder of the Rhapsody project).
- TargetProxyIPNumber
  This tag can be used to specify the IP number of the target proxy server.
- TargetProxyModelCodeSubFolder
  This tag can be used to specify the relative path from the model source code to the IDE project (see tag TargetProxyIDEProject). It is necessary only if this relative path is different than the relative path from the model code to the main code generation folder of the Rhapsody project (for example if the generated model code is deployed into a different folder hierarchy before building the application).
  This tag is relevant only when computing code coverage information.
- TargetProxyPortNumber
  This tag can be used to specify the port number of the target proxy server.
- TargetProxyStartupWaitCode
  This tag can be used to specify the code which is used after the application has started to wait before starting the test case.
- TargetProxyTimeoutBuild
  This tag can be used to specify a timeout when performing “Build TestCase|TestContext|TestPackage”.
- TargetProxyTimeoutExecute
  This tag can be used to specify a timeout for the test execution.

- rtc_assert_handling
  This tag must be set to by_id. This value reduces the amount of memory needed to store information for assertions during test execution.

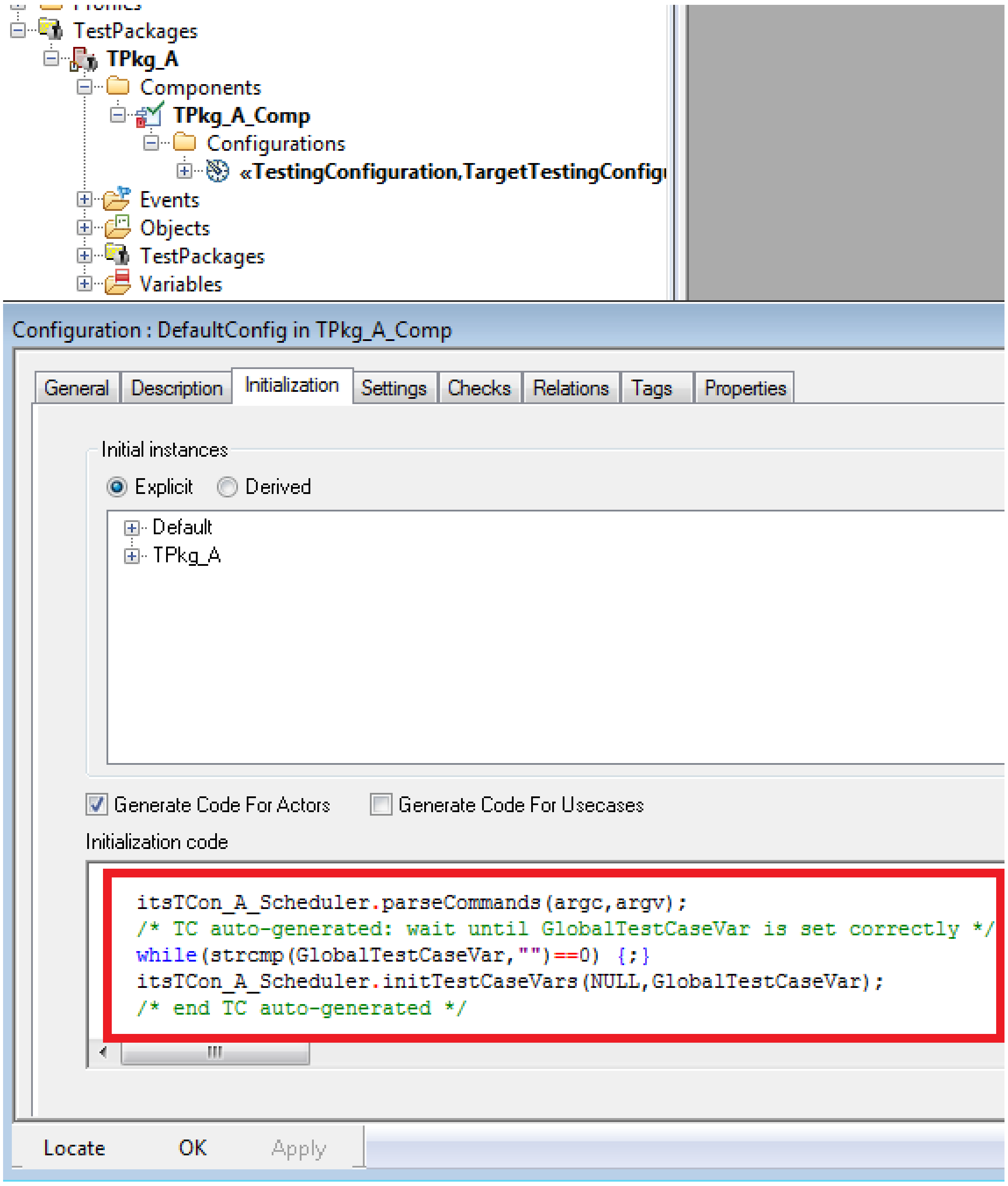
Note: If an update of the tests was already performed without the stereotype <<TargetTestingConfiguration>> set on the code generation configuration, the Initialization Code of the configuration contains the statement
itsTCon_A_Scheduler.parseCommands(argc,argv);

This statement (see Figure 4) needs to be removed manually by the user before tests can be executed on a small target because it causes errors when running tests on a small target. TestConductor automatically adds the lines
while(strcmp(GlobalTestCaseVar,"")==0) {;}
itsTCon_A_Scheduler.initTestCaseVars(NULL,GlobalTestCaseVar);

when updating the tests with <<TargetTestingConfiguration>> set on the code generation configuration but it does not remove the line
itsTCon_A_Scheduler.parseCommands(argc,argv), this is a limitation.

When removing the <<TargetTestingConfiguration>> to execute tests on the host again, the lines
while(strcmp(GlobalTestCaseVar,"")==0) {;}
itsTCon_A_Scheduler.initTestCaseVars(NULL,GlobalTestCaseVar);

need to be removed manually again.



Initial instances: [✓ Explicit] Derived

[✓] Generate Code For Actors [ ] Generate Code For Usecases

Initialization code

```
itsTCon_A_Scheduler.parseCommands(argc,argv);
/* TC auto-generated: wait until GlobalTestCaseVar is set correctly */
while(strcmp(GlobalTestCaseVar,"")==0) {;}
itsTCon_A_Scheduler.initTestCaseVars(NULL,GlobalTestCaseVar);
/* end TC auto-generated */
```

Figure 4: Initialization code of code generation configuration containing superfluous statements

Executing a TestCase

- Set the <<TargetTestingConfiguration>> (or a derived) stereotype manually on the CG Configuration
- Configure the additional TargetTestingConfiguration tags
- Update the TestCase (from the context menu of the TestCase)
- Start the target proxy server
- Switch on and prepare the target
- Build the TestCase (from the context menu of the TestCase). The following sequence takes place during the build:
    - "OpenIDEProject" TCP/IP command is send to the target proxy server with an empty IDE project path → Close IDE Project
    - Code generation
    - "OpenIDEProject" TCP/IP command is send to the target proxy server with an IDE project path → Open IDE Project
    - "Compile" TCP/IP command is send to the target proxy server
    - "FlashTarget" TCP/IP command is send to the target proxy server
- Execute the TestCase (from the context menu of the TestCase). The following sequence takes place during the execution:
    - "RunTarget" TCP/IP command is send to the target proxy server
    - "ExecuteTest" TCP/IP command is send to the target proxy server
    - Result verification and generation of a HTML result report

Computation of code coverage

Support for computation of code coverage while executing test cases on a target depends mainly on the compiler being used; some target specific limitations may apply. Supported compilers are Microsoft, GNU, ARM; not all compiler features and options are supported.

Some manual preparations must be done before computing code coverage (depending on the compiler, target and IDE being used some of these steps can be performed automatically).

Target configuration

For computation of code coverage, the source code needs to be instrumented (annotated) with some macros which are used to track the executed functions, statements, decision branches. For a correct annotation TestConductor needs some information about the compiler and the target system, the so called target configuration: The size and sign of some types, the endian of the target, the compiler family and some more. To collect this information a small program

must be compiled (with the same compiler and compiler options which are used to compile the tested application) and executed on the same target the tested application will be executed on. The target configuration tool will collect the needed information and write it into an xml file, this file can be used from then on for the instrumentation for code coverage until the target configuration (compiler, compiler options, target) changes. If the target does not support file I/O, the data can be printed to a console window or retrieved from application memory using a debugger.
If the application is tested directly on the host, TestConductor can collect the information about the target configuration automatically by building and executing the tool during annotation of the source code. This is supported for Microsoft or GNU compilers.

To manually build and execute the target configuration tool a Rhapsody model can be used which is part of the TestConductor installation, in folder TestConductor/CodeCoverage/TargetConfiguration. Copy the folder TargetConfiguration to a folder which can be written to and open the project in Rhapsody in C++ or C. The project contains one Code Generation Component for C++ and one for C, each with several configurations for different environments. If the project does not contain a configuration for the environment being used, create a new configuration or use an existing one and apply the necessary settings (like environment, compiler options). Otherwise, the source code file of the target configuration tool can be directly compiled outside of Rhapsody: The C and C++ version of the source code is located in folder TargetConfiguration/src.
Compilation of the tool can (and must) be controlled by some macros:

- TC_MAIN
  Can be used to define the code name of the main function of the target configuration tool (for example, if an external *main* function in another file is used).
  Default: main
  For some targets/compilers: vxmain

- TC_MAIN_WITH_VOID_ARGS
  Can be used to define the code name of the main function of the target configuration tool. If undefined or defined 0, the “main” function of the target configuration tool has signature *(int argc, int** argv)*. Otherwise the signature is *(void)*.
  Default: 0
  For some targets/compilers: 1

- TC_MAIN_WITH_VOID_RETURN
  Can be used to define the return type of the main function of the target configuration tool. If undefined or defined 0, the “main” function of the target configuration tool has return type *void*. Otherwise the return type is *int*.
  Default: 0
  For some targets/compilers: 1

- TC_XML_OUTPUT_KIND
  Can be used to control how to provide the collected information after executing the target configuration tool. If undefined or defined 0, the information is written into an xml file. If defined 1, the data is written to stdout (in xml format). If defined 2, the data is stored in the application memory (in xml format, variable name is xml_out_var).
  Default: <undefined>

- TC_SIZE_OF_XML_OUT_VAR
  Can be used to define the size of the variable to hold the collected information if a variable should be used (see TC_XML_OUTPUT_KIND).
  Default (if TC_XML_OUTPUT_KIND is undefined or defined 0): <undefined>
  If TC_XML_OUTPUT_KIND is defined 1 or 2: 4095

- TC_ERR_OUTPUT_KIND
  Can be used to control if error messages should be printed to stderr or stored in a variable. If undefined or defined 0, error messages are printed to stderr. If defined 1, error messages are stored in a variable (variable name is err_out_var).
  Default: <undefined>

- TC_SIZE_OF_ERR_OUT_VAR
  Can be used define the size of the variable to hold the error messages if error messages should be stored in a variable (see TC_ERR_OUTPUT_KIND).
  Default (if TC_ERR_OUTPUT_KIND is undefined or defined 0): <undefined>
  If TC_ERR_OUTPUT_KIND is defined 1: 140

- TC_ADDITIONAL_CODE1
  Can be used to provide additional code for the tool (like #include's or additional initialization code for the application).
  Default: <undefined>

- TC_ADDITIONAL_CODE2
  Can be used to provide additional code for the tool (like #include's or additional initialization code for the application).
  Default: <undefined>

After building the tool, load the tool binary on the target and execute it. It will collect the necessary data and write it to a file targetconf.xml if file I/O is supported. Otherwise the data will be printed to the console window or stored in the application memory: In this case the user must copy the data into an xml file.
The xml file with the information about the target configuration should be copied to a location which can be accessed during compilation of the application being tested, for example in the main or code generation folder of the application's project. The name of the xml file is arbitrary.

Options file for computation of code coverage

To compute code coverage, the user must provide some information about the installation of the compile environment in an xml options file. Below are two examples when using the Microsoft and ARM compiler. For details regarding compilers for VxWorks or Integrity refer to the corresponding TestConductor documents.

A template for an options file with comments explaining the available settings is provided in the TestConductor installation: Copy file **<RhapsodyInstall>**/TestConductor/TCCodeAnnotationOptions.xml to another location (for example, into the main folder of the Rhapsody project). Open the copy in an editor and enter the needed attributes and values in the <Environment> section:

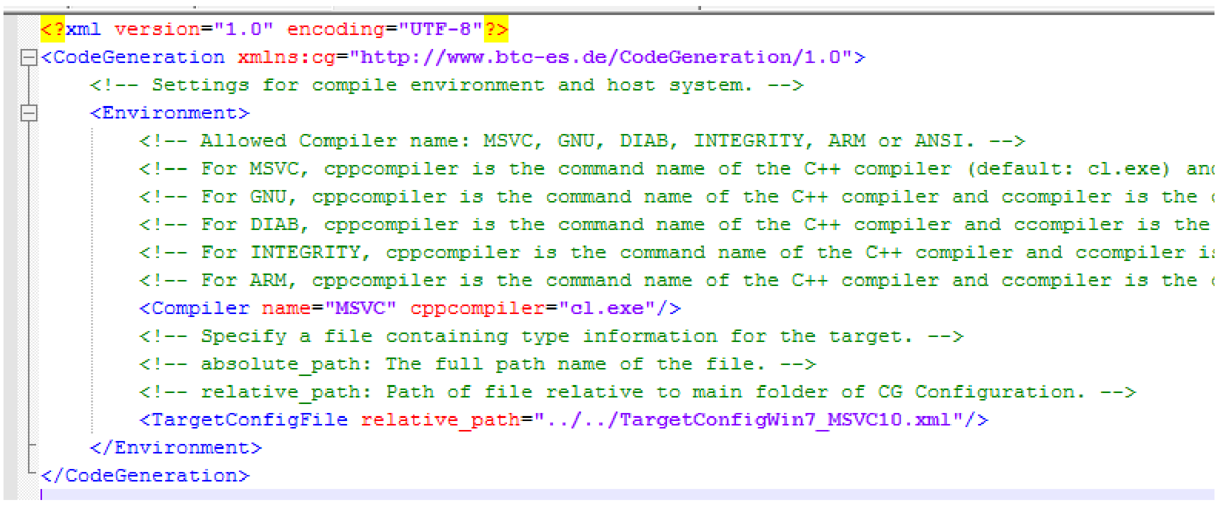
Example when using a Microsoft (MSVC) compiler:

- Attribute <Compiler>
    - name="MSVC"
    - cppcompiler= When using C++, enter the name of the Microsoft C++ compiler (example: cl.exe).

      Alternatively:
    - ccompiler= When using C, enter the name of the Microsoft C compiler (example: cl.exe).
- Attribute <TargetConfigFile>
    - relative_path= Enter the path and file name of the target configuration xml file, relative to the code generation folder.

      Alternatively:
    - absolute_path= Enter the full path of the target configuration xml file.

See figure 5 below for an example of an options file for computation of code coverage (MSVC compiler, C++).



```
<?xml version="1.0" encoding="UTF-8"?>
<CodeGeneration xmlns:cg="http://www.btc-es.de/CodeGeneration/1.0">
    <!-- Settings for compile environment and host system. -->
    <Environment>
        <!-- Allowed Compiler name: MSVC, GNU, DIAB, INTEGRITY, ARM or ANSI. -->
        <!-- For MSVC, cppcompiler is the command name of the C++ compiler (default: cl.exe) an
        <!-- For GNU, cppcompiler is the command name of the C++ compiler and ccompiler is the
        <!-- For DIAB, cppcompiler is the command name of the C++ compiler and ccompiler is the
        <!-- For INTEGRITY, cppcompiler is the command name of the C++ compiler and ccompiler i
        <!-- For ARM, cppcompiler is the command name of the C++ compiler and ccompiler is the
        <Compiler name="MSVC" cppcompiler="cl.exe"/>
        <!-- Specify a file containing type information for the target. -->
        <!-- absolute_path: The full path name of the file. -->
        <!-- relative_path: Path of file relative to main folder of CG Configuration. -->
        <TargetConfigFile relative_path="../../TargetConfigWin7_MSVC10.xml"/>
    </Environment>
</CodeGeneration>
```

Figure 5: Code coverage options to provide information about the MSVC compiler environment (example for C++)

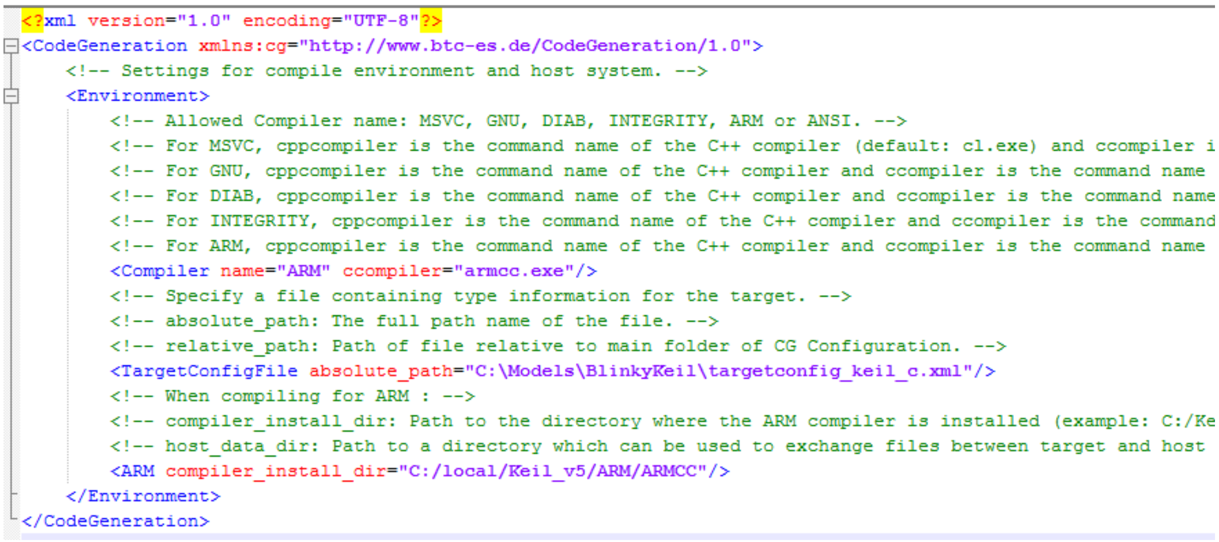
Example when using an ARM compiler:

- Attribute <Compiler>

    - name=”ARM”
    - cppcompiler= When using C++, enter the name of the ARM C++ compiler (example: armcc.exe).
    - ccompiler= When using C, enter the name of the ARM C compiler (example: armcc.exe).
- Attribute <TargetConfigFile>
    - relative_path= Enter the path and file name of the target configuration xml file, relative to the code generation folder.

      Alternatively:
    - absolute_path= Enter the full path of the target configuration xml file.
- Attribute <ARM>
    - compiler_install_dir= Enter the path to the folder where the ARM compiler is installed (example:
      C:/tools/Keil_v5/ARM/ARMCC.)

See figure 6 below for an example of an options file for computation of code coverage (ARM compiler, C).



```
<?xml version="1.0" encoding="UTF-8"?>
<CodeGeneration xmlns:cg="http://www.btc-es.de/CodeGeneration/1.0">
    <!-- Settings for compile environment and host system. -->
    <Environment>
        <!-- Allowed Compiler name: MSVC, GNU, DIAB, INTEGRITY, ARM or ANSI. -->
        <!-- For MSVC, cppcompiler is the command name of the C++ compiler (default: cl.exe) and ccompiler i
        <!-- For GNU, cppcompiler is the command name of the C++ compiler and ccompiler is the command name
        <!-- For DIAB, cppcompiler is the command name of the C++ compiler and ccompiler is the command name
        <!-- For INTEGRITY, cppcompiler is the command name of the C++ compiler and ccompiler is the command
        <!-- For ARM, cppcompiler is the command name of the C++ compiler and ccompiler is the command name
        <Compiler name="ARM" ccompiler="armcc.exe"/>
        <!-- Specify a file containing type information for the target. -->
        <!-- absolute_path: The full path name of the file. -->
        <!-- relative_path: Path of file relative to main folder of CG Configuration. -->
        <TargetConfigFile absolute_path="C:\Models\BlinkyKeil\targetconfig_keil_c.xml"/>
        <!-- When compiling for ARM : -->
        <!-- compiler_install_dir: Path to the directory where the ARM compiler is installed (example: C:/Ke
        <!-- host_data_dir: Path to a directory which can be used to exchange files between target and host
        <ARM compiler_install_dir="C:/local/Keil_v5/ARM/ARMCC"/>
    </Environment>
</CodeGeneration>
```

Figure 6: Code coverage options to provide information about the ARM environment (example for C)

In the Rhapsody model, use a tag of the Code Generation Configuration to specify the path to the options file: Open the feature dialog of the Code Generation Configuration and go to the Tags section. Then enter the path (including name and extension) to the options file into the tag CodeCoverageOptionsFilename. You can use an absolute path or a path relative to the code generation main folder (location of the Makefile).

Building and executing tests with computation of code coverage

To be able to compute code coverage information during test execution the source code needs to be instrumented (annotated). This is done by a tool which is part of the TestConductor installation. When building the application from Rhapsody the code is annotated in the background during the build process, when building the application in another IDE the user must ensure the annotation tool is called before the code is compiled (for example, by editing the IDE project settings or by manually performing the necessary steps).

Basically, two steps are necessary: Before compilation, the code must be annotated (by calling the tool TCCodeAnnotation.exe; this tool creates a copy of the not annotated source code file). After the annotated code has been compiled, rename the annotated version of the file and rename the not annotated file to its original name (to avoid errors when compiling the file another time). These steps can be entered directly in the IDE project settings (if the IDE supports defining pre or post build steps) or using batch files which are called manually. Below is an example how to create the needed batch files:

1. Create a file annotate.bat (for example in the project main folder or in the code generation folder) with this content (in one line):
   <RhapsodyInstall>/TestConductor/TCCodeAnnotation.exe **<CGPath> <Includes> <Defines> <ImplFileName>**

   <RhapsodyInstall> The Rhapsody installation path.
   <CGPath> The absolute path to the generated code (usually the location of the Makefile).
   <Includes> The list of all include directives to compile the code (can be copied for example from the Makefile or from the IDE project settings).
   <Defines> The list of all defined macros to compile the code (can be copied for example from the Makefile or from the IDE project settings).
   <ImplFileName> The file name and extension of the implementation file of the SUT.
   This batch file is used to call the TestConductor annotation tool before compilation of the implementation file.

2. Create a file copyfile.bat in the same folder with this content (in two lines):
   copy **<FullImplFile> <FullImplFile>**.annotated
   move **<FullImplFile>**.bak **<FullImplFile>**

   <FullImplFile> The full path and extension of the implementation file of the SUT.
   This batch file is used after compilation to rename the annotated implementation file and to move the backup of the original implementation to it's original name.

After providing these information, update and build the TestConductor tests. After building the application the tests can be executed on the target to compute code coverage information. After the execution of the tests has finished, TestConductor automatically adds a detailed code coverage report to the Rhapsody model.

Limitations

Automation of testing on a small target

- Commands like “Compile” or FlashTarget” or “RunTarget” can only be automated if the IDE (like µVision, IAT, Trace32) provides a suitable API. Otherwise the user is informed (by TestConductor or the target server proxy) and must do these steps manually.

Support of testing on a small target

- The animation based testing mode is not supported for the testing on a small target with TestConductor.

Eclipse Target Proxy

Scope

The Eclipse Target Proxy is a plug-in for Eclipse 4.2.2 (Juno). It provides a simple server that can be contacted via TCP/IP and that allows Rhapsody to communicate with Eclipse in order to trigger the “Build” and “Debug” process of generated code in C and C++. Note that the Eclipse Target Proxy is intended for demonstration purposes and thus provides only a brief example of how to implement testing on a small target with TestConductor.

Prerequisites

The plug-in requires some software to be running on your system. The following products are mandatory:

- Cygwin 1.7.25 or higher
- gcc 4.7.3 or higher
- GDB 7.4 or higher
- Eclipse 4.2.2 (Juno) or higher

Cygwin serves as an execution environment that both, TestConductor and Eclipse CDT are compatible with. GCC is the compiler of choice and GDB is a debugging environment that the Eclipse CDT is compliant with. Alternatively, other compilers can be used but in this case it might be necessary to install other Eclipse plugins and to adjust the *Eclipse Target Proxy* to use the API of the used compiler and debugger.

It is recommended to use the “Eclipse IDE for C/C++ Developers” Package from the Eclipse website as it contains all plug-ins that the *Eclipse Target Proxy* is depending on. However, if you want to use your own configuration of Eclipse, the following table shows all plug-in dependencies of the *Eclipse Target Proxy*:

Plug-In ID	**Minimum Version**
org.eclipse.ui	3.104.0
org.eclipse.swt	3.100.1
org.eclipse.swt.win32.win32.x86	3.100.1
org.eclipse.jface	3.8.102
org.eclipse.core.commands	3.6.2

Plug-In ID	**Minimum Version**
org.eclipse.ui.workbench	3.104.0
org.eclipse.e4.ui.workbench3	0.12.0
org.eclipse.core.runtime	3.8.0
org.eclipse.osgi	3.8.2
org.eclipse.equinox.common	3.6.100
org.eclipse.core.jobs	3.5.300
org.eclipse.core.runtime.compatibility.registry	3.5.101
org.eclipse.equinox.registry	3.5.200
org.eclipse.equinox.preferences	3.5.1
org.eclipse.core.contenttype	3.4.200
org.eclipse.equinox.app	1.3.100
org.eclipse.cdt.debug.core	7.2.0
org.eclipse.debug.ui	3.8.2
org.eclipse.debug.core	3.7.100
org.eclipse.core.resources	3.8.1
org.eclipse.cdt.dsf.gdb	4.1.1
org.eclipse.cdt.dsf	2.3.0
org.eclipse.cdt.core	5.4.1
org.eclipse.cdt.launch	7.1.0

Installation of the Plugin

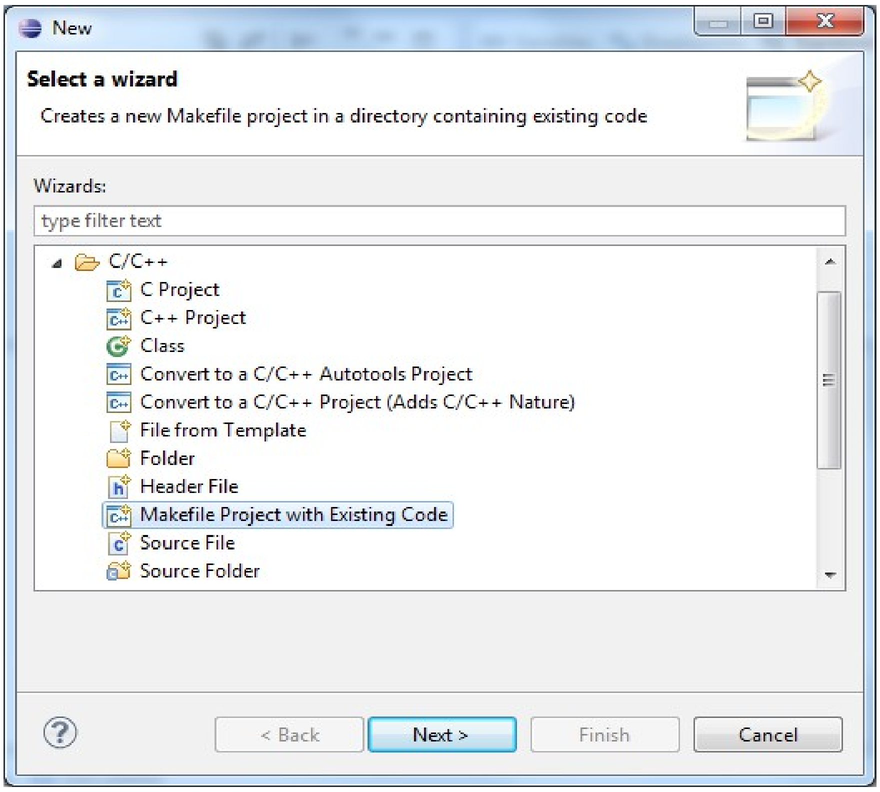
In order to install the Eclipse Target Proxy, simply copy the EclipseTargetProxy.jar from **TestConductor\EclipseTargetProxy\lib** within Rhapsody's user profile folder to the "plugin" directory within your Eclipse installation path and restart Eclipse in case it was already running.

You can check whether the plug-in is running by running Eclipse and selecting *Help → About Eclipse → Installation Details*. The plug-in should appear in the *Plug-Ins* tab.

Preparing a project in Eclipse

To set up your Eclipse project, you first need to generate the code for your TestComponent, so Eclipse is able to create a workspace project from it. Make sure, that the makefile of your project is called “Makefile” without a file extension, so Eclipse will be able to find it without further adjustment. You can adjust the name and extension of your makefile by configuring the properties *CPP_CG::Cygwin::MakeExtension* and *CPP_CG::Cygwin::MakeFileName* or *C_CG::Cygwin::MakeExtension* and *C_CG::Cygwin::MakeFileName* for C++ and C projects, respectively, in Rhapsody.

After the code has been generated, you can invoke the Eclipse Project Wizard by selecting *File → New → Project...* from the Eclipse menu. Choose *Makefile Project with Existing Code* from the *C/C++* Category as a project type as shown in Figure 7 below.


New

Select a wizard

Creates a new Makefile project in a directory containing existing code

Wizards:

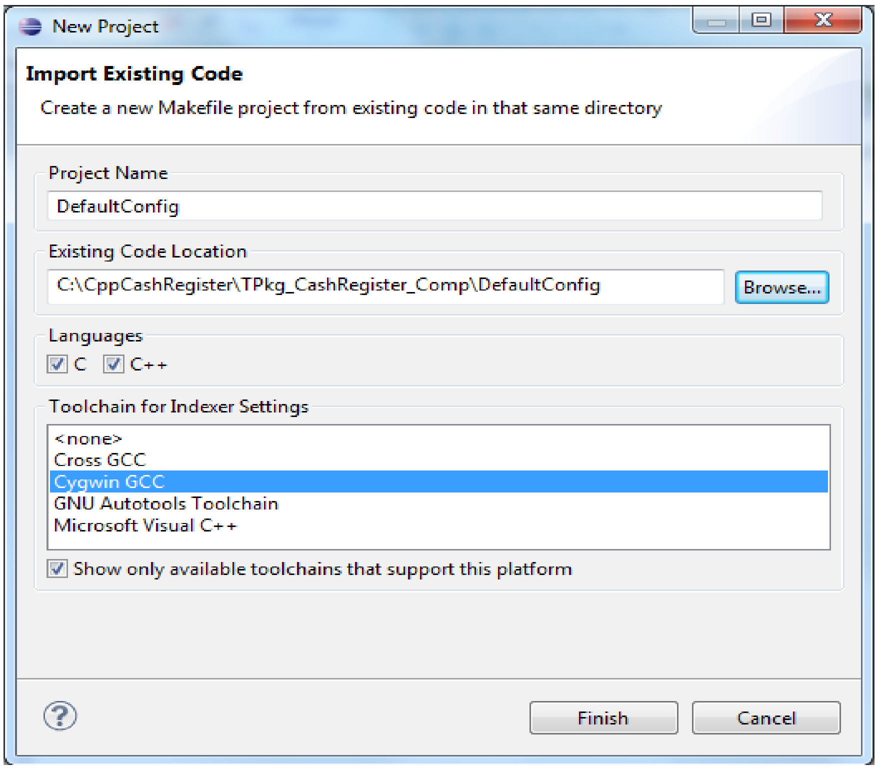
type filter text

- C/C++
  - C Project
  - C++ Project
  - Class
  - Convert to a C/C++ Autotools Project
  - Convert to a C/C++ Project (Adds C/C++ Nature)
  - File from Template
  - Folder
  - Header File
  - Makefile Project with Existing Code
  - Source File
  - Source Folder

< Back | Next > | Finish | Cancel

Figure 7: Eclipse Project Wizard

Figure 8 shows the emerging window. You can assign a name to your project and specify the location of the generated code. It is mandatory that the code location specified here matches not only the location of your generated code in the file system, but also the tag *TestArchitecture::TargetTestingConfiguration::TargetProxyIDEProject* in Rhapsody. This will enable Eclipse to determine the right project for specific TestConductor commands sent by Rhapsody. The project's name, on the other hand, can be chosen arbitrarily. The rest of the settings in this dialog should match the settings given in Figure 8.


New Project

Import Existing Code

Create a new Makefile project from existing code in that same directory

Project Name: DefaultConfig

Existing Code Location: C:\CppCashRegister\TPkg_CashRegister_Comp\DefaultConfig

Languages: ☑ C ☑ C++

Toolchain for Indexer Settings:
- <none>
- Cross GCC
- Cygwin GCC
- GNU Autotools Toolchain
- Microsoft Visual C++

☑ Show only available toolchains that support this platform

Figure 8: Creating a Makefile Project

After closing this dialog by clicking *Finish*, Eclipse will generate the project for you. You can find it in the *Project Explorer* and now access it from TestConductor.

Usage

Once your Eclipse project is set up, you can start accessing it from remote from within TestConductor. To do so, make sure that the code generation Configuration has a stereotype which inherits from stereotype <<TargetTestingConfiguration>>. In the Features view of your Configuration, set up your connection parameters such as IP address and port number. The default port number of the *Eclipse Target Proxy* is 6789.

For convenience, the TestConductor Testing Profile contains a predefined stereotype <<ETPTargetTestingConfiguration>> which already provides correct values for most of the settings when using the *Eclipse Target Proxy*. For example, when using this stereotype for functionality like exiting the tested application or storing assert information the default implementation is used. In this case, the macro "TC_SMALL_TARGETS_ALLOW_DEFAULT_IMPL" has to be defined for the code generation configuration.

To build a *TestPackage*, *TestContext* or *TestCase*, you can simply use the standard commands from the context menu. TestConductor will automatically send these commands to your *Eclipse Target Proxy*. However, make sure that you always build your *TestPackage*, *TestContext* or *TestCase* before executing it to forward any changes to Eclipse.

Be aware, that the *Eclipse Target Proxy* is not an actual target that needs to be flashed. The *FlashTarget* command from TestConductor is, hence, not supported. Therefore, you can ignore the corresponding warning given by TestCondcutor after the build process.

The *Eclipse Target Proxy* supports computation of code coverage while executing tests. When using MSVC, Cygwin or Linux compile environments no additional settings are necessary and the target configuration and annotation can be processed in the background during code compilation. When using other compilers (or compile environments with non default names) the user must provide additional information to be able to compute code coverage. See Computation of code coverage for details.

Developer Notes

Although the *Eclipse Target Proxy* is meant for demonstration purposes, it may form the basis for a more complex target proxy. Hence, this section gives an overview over the most important Java classes the plug-in consists of. The source files can be found in **TestConductor\EclipseTargetProxy\src** within Rhapsody's user profile folder.

ProxyServer.java

This class opens the socket for TestConductor and receives all incoming commands. These commands are handled in dedicated functions for opening the IDE, opening the project, building the generated code, flashing the target (which is currently ignored as there is no target), running the target (which is ignored for the same reason) and executing (debugging) the project. In order to implement own build, flash and debug processes here, you can simply edit these functions to call your own classes.

TC2EGdbLaunchDelegate.java

This is a subclass of GdbLaunchDelegate, which is responsible for setting up the debugging environment. It is needed to invoke the creation of a TC2EServicesFactory.

TC2EServicesFactory.java

This class is a subclass of GdbDebugServicesFactory starts the specific services needed for debugging. It creates a custom RunControlService, TC2EGDBRunControl.

TC2EGDBRunControl.java

The TC2EGDBRunControl service controls the state of the debugger. In the plug-in, it is used to apply a listener to the CommandControl service of GDB, that forwards all commands coming from Eclipse to GDB. This enables the plug-in to determine when the breakpoint at the main-function is reached and the name of the TestCase to run can be set, again using the CommandControl service to change the value of the global variable *GlobalTestCaseVar*.

ITC2EPluginConstats.java

This interface holds important constants of the plug-in, such as the port number of the server and the commands coming from TestConductor as well as the expected answers.