

Building C++ Applications with RSARTE

Mattias Mohlin
Senior Software Architect
IBM

INTRODUCTION **3**

TRANSFORMATION CONFIGURATIONS **4**

Transformation Configuration Properties 5
Main tab *6*
Sources and Target tab *6*
Properties tab *7*
Target Configuration tab *10*
Threads tab *13*
Marking Models tab *14*
Extensions tab *14*
Use of Variables in Transformation Configurations 15
User-defined Variables *15*
Creating Transformation Configurations 16
Inheritance and Abstract Transformation Configurations 17
Prerequisite Transformation Configurations 19
Active Transformation Configurations 19
Managing the Sources of a Transformation Configuration 20
Updating Sources Using Ant Task *21*
Detect Source Dependencies Automatically *22*
Context Sensitive Library Builds *23*
Automated Source Management and External Code *24*

BUILDING GENERATED CODE **24**

Makefile Generation 26
Two Types of Builds – Model Compiler versus Classic Builder 26

INTERACTIVE BUILD **27**

Build Messages 28
Building Multiple Transformation Configurations 30
Alternative Ways to Trigger an Interactive Build 31

BATCH BUILD **32**

Model Compiler 32
Headless RSARTE 32
Building with an Ant Build File *33*
Building Directly from the Command-Line *36*

BUILD VARIANTS **36**

Build Variant and Transformation Configuration Framework APIs 41

EXTERNAL LIBRARIES....41
PRECOMPILED LIBRARIES....42
EXTERNAL CONSTANTS....43
Defining External Constants Programmatically....44
CLEAN....45

This document describes the steps involved when building a real-time C++ application from a UML-RT model in Rational Software Architect Real-Time Edition (RSARTE).

Readers of this document are assumed to have read the document "Modeling Real-Time Applications in RSARTE".

The document was last updated for RSARTE 10.2. All screen shots were captured on the Windows platform.

Introduction

One of the most important capabilities of RSARTE is the ability to transform a UML model into an executable real-time application. This process, which we refer to as "building the model", typically consists of the following steps:

1. A subset of the model is transformed to C++ code.
2. A CDT project and a makefile is generated.
3. A make tool is launched to build the generated code using the makefile.
4. Messages (such as compilation errors) that are produced during the build are captured and printed.

There are two ways of building a model; **interactive build** from within the RSARTE user interface, and **batch build** from command line or scripts[1]. In both cases the build is done in the same way, by performing the steps mentioned above. The differences between interactive and batch builds are more related to how the build is triggered, and what happens after the build is done. For example, in an interactive build most build messages are printed to the UML Development Console while in a batch build they are typically printed to the command line console or written to a log file.

RSARTE provides two separate tools for building a model; the **classic builder** and the **model compiler**. The classic builder builds the model according to the four steps outlined above, where the code generation step has to run inside RSARTE. The main drawback with this approach is that during batch builds it is necessary to run RSARTE in headless mode in order to generate the code. This cannot be done on systems where the DISPLAY variable is not set. There is also a drawback for interactive builds because some parts of the RSARTE user interface cannot be used during code generation (all parts that may need to access the model files that are being transformed). Hence, with the classic builder you often have to wait for the code generation step to complete before you can proceed working with RSARTE.

The model compiler tool is a stand-alone command line tool which does not have these drawbacks. It can be used for true batch builds that run without any dependency on the RSARTE IDE. It can also be used for interactive builds from within the IDE. In that case RSARTE will launch the model compiler for generating the code and a make file, and then generated code is built as usual by make.

If your system supports parallel execution of make rules it’s also possible to generate a single make file from RSARTE that also contains rules that invoke the model compiler for the code generation. This means that the entire build can be driven by a single make file. Even if this could boost build performance by parallelizing the generation of C++ files, it also means that the model compiler will be invoked multiple times, which involves some overhead. You have to measure the performance on your system to decide if this approach is worthwhile or not.

In the following chapters we will go through different aspects that are related to building an RSARTE C++ model. Most of these are the same regardless if you choose to use the classic builder or the model compiler.

1 A third way is to trigger a build programmatically from a plugin using available APIs. These APIs are not covered in this document.

Transformation Configurations

The transformation of a model to compiled code can be done in many ways, and is controlled by several properties. Examples of such properties include

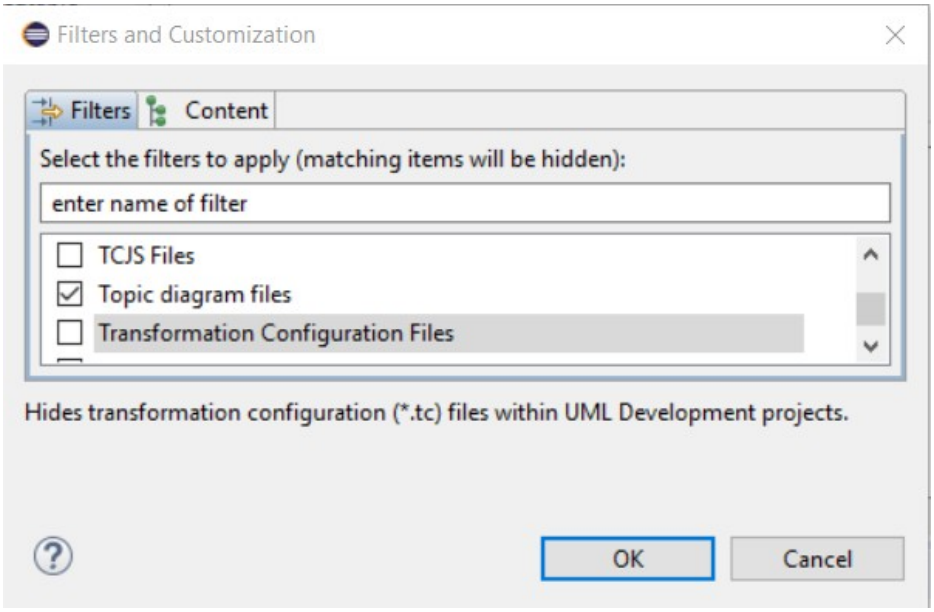
- what subset of the model should be built
- how should the generated C++ code be compiled
- which target configuration of the RT services library should be used
- etc.

All such properies that are used in a build are stored in a **transformation configuration** (or TC for short), which needs to be specified when starting the build. A TC is a text file, and can be in two different formats

1. Traditional format (*.tc)
   This format is supported both by the classic builder and the model compiler.
2. JavaScript format (*.tcjs)
   This format is supported by the model compiler only.

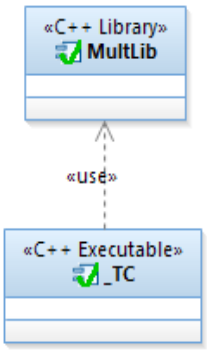
The new JavaScript TC format has many benefits over the traditional format. For example, it supports dynamic settings where the value is not a static value but computed by JavaScript code. It is also more readable than the traditional format and easy to edit with any kind of text editor. The traditional format is not recommended to edit in a text editor since it's easy to accidentally corrupt the file (due to complex string encodings used in that file format). RSARTE provides a dedicated TC editor that should be used instead. A TC editor is also provided for the JavaScript format, although for that format it's safe to edit the file in a regular text editor. Note that there are good text editors for JavaScript available for Eclipse that can be installed on top of RSARTE.

By default TC files are not shown in the Project Explorer, and instead TCs are shown as components under a virtual folder. This folder is called "Transformation Configurations" for TC files in the traditional format, and "TCJS Transformation Configurations" for TC files in the JavaScript format. To make the actual TC files visible you have to click on the view menu in the Project Explorer, and select "Filters and Customization". Turn off the filter called "Transformation Configuration Files" (for traditional format) or "TCJS Files" (for JavaScript format):



Note that in the “Content” tab it is also possible to hide or show the virtual folders "Transformation Configurations" or "TCJS Transformation Configurations".

The UML representation of a TC is a component. You can therefore visualize TCs by dragging them from the Project Explorer and dropping them on a component diagram. Certain properties of the TC can be represented as relationships in the component diagram, in order to show how a TC use or inherit from other TCs. Here is an example:



This diagram shows two TCs, "MultLib" and "_TC". The usage dependency corresponds to a setting "Prerequisite transformation configurations" and specifies that in order to build "_TC" we must first build "MultiLib".

This visualization of TCs in a component diagram is only available for TCs in the traditional format.

Transformation Configuration Properties

If you double-click on a TC file (or on a component that represents the TC) the transformation configuration editor will open. This editor lets you edit all the properties that are stored in the TC. The editor uses several tabs to group related settings. The TC editor for the JavaScript format uses slightly different tabs and in some cases shows the TC properties in a different or-

der or on different tabs than what is the case for a TC in the traditional format. In this chapter we will go through the TC properties as they appear in the TC editor for the traditional format. However, note that the TC properties are the same regardless of TC file format, and they have the same names in both TC editors, so the description of the TC properties apply for both TC formats.

Main tab

This tab is an overview of the TC, and does not contain any properties that affect the transformation as such. It shows the **Name** of the transformation ("UML to RealTime C++") and provides a “Run” button for running (i.e. building) the TC. You can use the **Documentation** field at the bottom for adding some documentation about the TC if you like to. The checkbox **Generate a debug log** can be used to produce a log file that shows details about what happens during the transformation. It can be useful when customizing a transform using your own plugin.

Sources and Target tab

This tab contains the **Sources** property that specify the subset of the model that will be transformed. These model elements constitute the source of the transform. Elements that are contained in a specified source element will also be transformed.
It can be convenient to specify a single top-level package as the source of the transform. Thereby the TC does not have to be updated when new elements are added to the package. However, this approach also means that some elements may be unnecessarily transformed, if the package contains elements that do not need to be transformed to C++. This will increase the time it takes to build the TC. See Managing the Sources of a Transformation Configuration for more information about different ways in which you can manage the list of source elements for a TC.

The property **Context sensitive library builds** can be set in order to optimize the build of prerequisite library TCs so that their lists of source elements are filtered to avoid building elements that are not referenced by the source elements of the built TC. Each library will hence be analyzed to determine what parts of the library that are necessary in the context of the built TC, and only those parts will be built. This can significantly speed-up the build of a TC with prerequisite TCs. See Context Sensitive Library Builds for more information about this feature.

The **Target** properties specify a CDT project to which generated C++ files will be written. This project can either be created and updated during the transformation, or you may use an existing project which you maintain manually. In the latter case the properties in the “Target Configuration” tab is not used, and corresponding properties from the target CDT project is used instead for building generated code. Note that if you go for the latter approach you do not have all build properties located in the TC, but some are then located in the CDT project. The default and the recommended choice is to automatically generate the CDT project from the TC. Read more about the "Automatically create and update target project" setting in Building Generated Code.

Note that the text you type in the "Workspace output path" field may include a path within the target CDT project. This makes it possible to put generated files in a subdirectory of the CDT

project. For example, the output path "/targetProj/src" means that generated C++ files will be placed in the folder "src" under the CDT project "targetProj".

If you choose to automatically create the target CDT project it will by default be placed in the workspace directory. You can change this default location using the “Location” field.

Properties tab

This tab contains all properties that are used when transforming the model to C++ code. It also contains properties that control how to generate the makefile to be used for building generated C++ code.

Top capsule is a property that is only available for a TC that builds an executable. It specifies which capsule that should be automatically incarnated when the executable is run. The top capsule is hence the entry point of the application. The top capsule is always a source element of the TC (explicitly or implicitly), or is located in one of the source elements.

Abstract specifies whether the TC is abstract or not. Abstract TCs only serve as a container for common properties that are shared between multiple TCs and they cannot be built themselves. See Inheritance and abstract transformation configurations for more information about abstract TCs.

Add Connexis support to build specifies if the generated code makes use of Connexis or not. This property is only available if you have installed the Connexis component of RSARTE.

Compile data classes individually controls how data classes (i.e. passive classes) are compiled. By default this property is true, which means that each data class will be built separately by a dedicated rule in the generated makefile. It can look something like this:

```
DataClass$(OBJ_EXT) : ../DataClass.cpp ../UnitName.h ../DataClass.h
    @$(FEEDBACK) Compiling Doc_target:DataClass
    $(1_CC) $(CC_HEAD) $(1_CCFLAGS) $(1_INCPATHS) ../DataClass.cpp $(CC_TAIL)
```

If you set this property to false, then make rules for data classes will not be generated, and instead they will be included at the end of the generated unit file (by default called UnitName.cpp):

```
#define PRAGMA_IMPLEMENTED
#include "DataClass.cpp"
```

The main reason for not compiling data classes individually would be to improve compilation time (compiling one big file can be faster than compiling several small files). However, if you use a build system that supports parallelization of make rules, then the opposite may be true.

Enable support for file artifacts is by default turned off. You can enable this property if your model contains artifact model elements with C++ code snippets. Such artifacts (called file artifacts) will be translated into C++ header and implementation files. A file artifact may contain two code snippets; a “Header” code snippet which is generated into the header file and an “Implementation” code snippet which is generated into the implementation file. You

can hence use file artifacts as a means for including custom hand-written C++ files in the generated application.

Generate qualifier for user's code is by default turned off. If it is turned on an extra line will be generated before each user code snippet. This line contains the fully qualified name of the UML element to which the code snippet belongs. This information can make it easier to understand the connection between generated C++ code and the model. It can also be used by 3rd party tools that analyze generated C++ code.

Here is an example of what a generated user code snippet could look like with this property turned on. The first line will not be generated when this property is off.

```
// ELEMENT: HelloWorld::HelloWorld::State Machine::Region1::Initial::Initial
//{{{USR platform:/resource/DoxygenTest/HelloWorld.emx#_xjldcPkAEeGEhK1G362qaA
log.log("Hello World from C++ Capsule");
context()->abort();
//}}}USR
```

Optimize handling of frequent triggers is an optimization property for generating C++ code that is more efficient in handling frequently triggered state machine triggers. If you know that some triggers in your capsules' state machines get triggered much more frequently than others, you can mark those triggers with the <<frequent>> keyword. When this property is set the `rtsBehavior` function will contain special if-statements for matching the current state, and the event and port of the current message, with the state, event and port of the frequent triggers. These if-statements are placed early in the `rtsBehavior` function to ensure that as little code as possible needs to execute when dispatching a message for a frequent trigger. Here is an example of what such an if-statement may look like:

```
if( /* frequent trigger */ LIKELY(stateIndex == 4/*Initial*/ && portIndex == 2/
*p*/ && signalIndex == PROTO::Base::rti_IE1) )
{
    chain3_e1();
    return;
}
```

Note that the code uses a macro LIKELY which is generated into the unit name header file. For GCC the macro is defined like this to give the compiler a hint that it's likely that the condition will be true:

```
#define LIKELY(x) __builtin_expect((x),1)
```

Artifact type specifies if generated C++ code should be compiled into an executable (default) or a library. There is also the choice of external library, which makes it possible to represent an external C++ library by means of a TC. This is useful in order to integrate the building of external code with building the model. See External Libraries for more information.

A special kind of executable TC is the “C++ Test Executable”. It works in the same way as a regular executable except that it does not specify a top capsule. This kind of TC is useful in order to build executables which run unit tests of model elements.

Note that some TC properties are only relevant for one particular artifact type. For example, a top capsule can only be specified for C++ executables.

Documentation comment style specifies what comment style to use for documentation comments in generated C++ code. A documentation comment is the text you can enter in the Documentation property tab when selecting an element in the model. By default C++ style comments (// ...) will be used. The code generator also supports two comment styles that can be used with the Doxygen publishing tool: Doxygen_JavaDoc (/** ... */) and Doxygen_QT (/*! .. */). Choose one of these if you plan to run Doxygen on generated code.

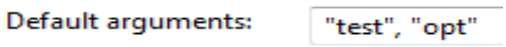
If documentation comments contain rich text (i.e. markup such as underlining, colors etc) they will be converted to plain text by the C++ code generator.

Environment is usually set to "C++ TargetRTS". However, if none of the generated C++ files will depend on anything from the RT services library you may instead set it to "C++".

Inherited transformation configurations lists other TCs from which this TC inherits. TC inheritance is a mechanism that allows you to group common properties in a single TC rather than to duplicate them into many TCs. See Inheritance and abstract transformation configurations for more information about transformation configuration inheritance.

Prerequisite transformation configurations lists other TCs which are "prerequisites" of this TC. See Prerequisite transformation configurations for more information.

Default arguments is a property that is only available for a TC that builds an executable or test executable. It specifies the default command-line arguments to use, in case the executable is started without providing any command-line arguments. The arguments should be a comma-separated list of valid C++ strings. Here is an example:



Output subdirectory is usually left empty. However, if you target the same CDT project from multiple TCs you may want to place the code that is generated from each TC into a separate subdirectory, in order to avoid naming conflicts between generated files and to make the structure of the CDT project more clear.

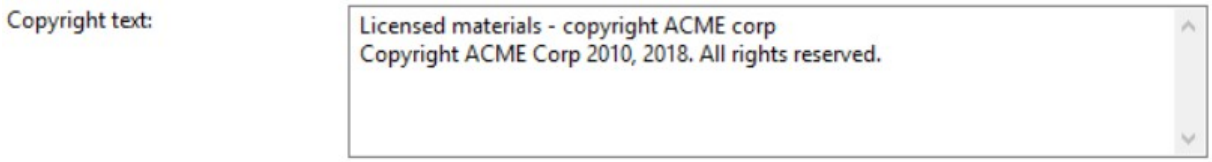
Unit name specifies the base name of the "unit" files. By default this property is "UnitName" which means that the unit files will be called "UnitName.cpp" and “UnitName.h”. The unit files contain certain information that applies to the whole unit of code that is generated from a TC. For example, you will find the mapping of logical threads to physical threads in “UnitName.cpp”. The `RTMain::entryPoint()` function, which is the generated application's entry point, is also located there.
The unit header file is included in each generated implementation file.

Unit subdirectory is usually left empty, but in case you want the unit file to be generated into a specific subdirectory you can specify it here. One reason for using this property could be that the specified unit file name clashes with the name of another generated file.

Include unit header file without subdirectory path can be set to just use the name of the header unit file in #include directives, without the unit subdirectory path. The unit header file is included in each generated C++ file, and by default the #include directive looks like this:
```
#include <unit-sub-dir/UnitName.h>
```
However, if this property is set the unit subdirectory path is omitted, and the #include directive will instead be:
```
#include <UnitName.h>
```
This is useful in case the preprocessor include path contains the unit subdirectory.

Common preface allows you to write some code that will be inserted verbatim into the header unit file (by default called "UnitName.h"). Since the header unit file is included by all files that are generated from the model, you can use the common preface to define or include definitions that should be available everywhere in generated code.

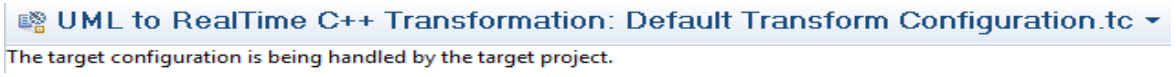
Copyright text may be used to insert a common comment block in the beginning of each generated file, for example a copyright text. Here is an example:




```
// Licensed materials - copyright ACME corp
// Copyright ACME Corp 2010, 2018. All rights reserved.
```

Target Configuration tab

This tab contains most of the properties that control how to generate the makefile to be used for building generated C++ code. If the “Target” properties in the “Sources and Target” tab specify an existing CDT project to use for building generated C++ files, then the properties in this tab are not applicable since the properties in the CDT project will be used instead. In this case you will see the following message in this tab:



Also, if the “Artifact type” property in the “Properties” tab is set to “C++ External Library” then this tab will contain different properties as described in External Libraries.

Use absolute paths in generated makefile should normally be unset, but if set the generated makefile will use absolute rather than relative paths for some of its variables.

Make type specifies the dialect of the generated makefile. The following dialects are supported:

- Microsoft (MS_nmake)
- Unix (Unix_make)
- GNU (Gnu_make)
- ClearCase (ClearCase_clearmake or Clearcase_omake)

By default this property is <default> which means that the makefile dialect will be determined based on the OS that is used (MS_nmake for Windows, and Unix_make for Unix).

Compile arguments specifies additional arguments to use when compiling generated C++ code. For example, if you want to add debug information to compiled code you can specify the `-g` flag if using the GNU C++ compiler.

Compile command is by default set to $(CC). This variable expands to the compiler to use, which follows from the property "TargetRTS configuration". If you want to use a different compiler than the one that is by default used for the selected TargetRTS configuration, then you can specify which compiler to use through this property.

Executable name specifies the name of the generated executable. This property is only available for executable TCs, and it is by default set to executable$(EXEC_EXT). The variable $(EXEC_EXT) expands to the file extension to be used for executable files on the target platform (which follows from the property "TargetRTS configuration").

Library name specifies the name of the generated library. This property is only available for library TCs, and it is by default set to library$(LIB_EXT). The variable $(LIB_EXT) expands to the file extension to be used for library files on the target platform (which follows from the property "TargetRTS configuration").

Inclusion paths allows you to specify additional include paths for the C++ compiler. Type each include path on a separate line. For example:



Note that you do not have to explicitly specify inclusion paths for the current target project or any of the prerequisite target projects, because these are added automatically by the makefile generator. For example, assuming that the above two inclusion paths were specified for a TC with a prerequisite TC ”libTC”, then the generated makefile would have the following inclusion paths:

```
0_INCPATHS = \
      $(INCLUDE_TAG)../../libTC_target \
      $(INCLUDE_TAG).. \
      $(INCLUDE_TAG)C:/mylibs \
      $(INCLUDE_TAG)C:/path
```

The variable $(INCLUDE_TAG) expands to the compiler flag to use for specifying include paths (e.g. `-I`).

Link command specifies the link command to use. This property is only available for executable TCs, and is by default set to $(LD). This variable expands to the linker to use, which follows from the property "TargetRTS configuration". If you want to use a different linker than the one that is by default used for the selected TargetRTS configuration, then you can specify which linker to use through this property.

Link order (custom) provides a way to control the link order for libraries when linking an executable. The default link order is specified in a workspace preference (*UML Development - Real Time C++ Transformations – Link Order*) and is by default:

```
$(USER_LDFLAGS) $(ALL_OBJS_LIST) $(USER_OBJS_LIST) $(UNIT_LIBS) $
(USER_LIBS)
```

You may rearrange these variables if you need libraries to be linked in another order. Any value set for this TC property overrides the value of the workspace preference for that particular TC. Hence, if you want a particular link order to be used for all your TCs, you may instead change the value of the workspace preference.

Build library command specifies the command for building libraries. This property is only available for library TCs, and is by default set to $(AR_CMD). This variable expands to the command to use for creating a library, which follows from the property "TargetRTS configuration".

Build library arguments specifies arguments for the build library command. This property is only available for library TCs.

Make command and **Make arguments** specify which make command and arguments to use. The make command is by default $defaultMakeCommand which expands to the name of the make tool to use, which follows from the property “TargetRTS configuration”. By default the flag `-s` is used (silent make, without echoes). Separate the make arguments using a space.

Target configuration name maps to a folder in the target CDT project where all generated files that are not source code will be placed. This includes for example makefiles and the files that are produced by these makefiles (typically binaries). The property is by default set to “default”. If you target the same CDT project from multiple TCs you must ensure that all these TCs have different target configuration names to avoid file name clashes and accidentally overwriting files generated by one TC with files generated by another TC.

Target services library specifies the location of the RT services library to use. The default value of this setting is ${RSA_RT_HOME}/C++/TargetRTS which points at the folder in the RSARTE installation where the C++ version of the RT services library resides.

If you have imported your TC from Rose RT, the RT services library in Rose RT will be used instead. If you prefer to use the implementation in RSARTE instead, which is somewhat improved, you should therefore update this setting after the import from Rose RT.

TargetRTS configuration specifies which target configuration to use. A target configuration is a specific version of the RT services library that is adapted to the specific target environment that is used. Read more about target configurations in the document “RT Services Library”. The target configurations that are shown in the drop down menu for this property are dynamically extracted from the specified “Target services library” directory. Hence, if you don't see any target configurations in the drop down menu, ensure that you have set-up “Target services library” to point to a valid directory that contains target configurations for the RT services library.

Top make arguments and **Top make command** specify the make command and arguments to use for the invocation of the top-level make file (called batch.mk). You may use these properties to execute some "pre-make" commands before the real build starts, for example by using a script as the top make command. It is only useful to set these properties if the “Batch Make Type” preference is set to Recursive, because then the top-level make file will contain recursive calls to other make files. If these properties are empty, the properties “Make command” and “Make arguments” will be used instead. See Makefile Generation for more information about inclusive and recursive makefiles.

User libraries allows you to specify custom libraries to pass to the linker. Type each user library file on a separate line.

Compilation make insert can be used to insert custom contents into the generated makefile. The text that you enter in this field will be copied verbatim into the generated makefile, just before the make rules section. You can use this property to add user-defined rules, variables, directives etc. to the makefile. For more information about what variables that are available to use in the compilation make insert fragment refer to the generated makefile. Also see the file `<RSARTE INSTALLATION>/rsa_rt/C++/TargetRTS/libset/default.mk`.

Link arguments specifies additional arguments for the linker. Type each link argument on a separate line.

User object files allows you to specify custom object files to pass to the linker. Type each object file on a separate line.

Threads tab

This tab controls the mapping of logical threads onto physical threads. See the document “RT Services Library” for more information about logical and physical threads.

By default there are two physical threads:

- **MainThread**
  This is the thread which by default runs all capsule instances in the model.
- **TimerThread**
  This thread is used by the Timing service of the RT services library to implement timers. It is always present even if you don't use timers in the model.

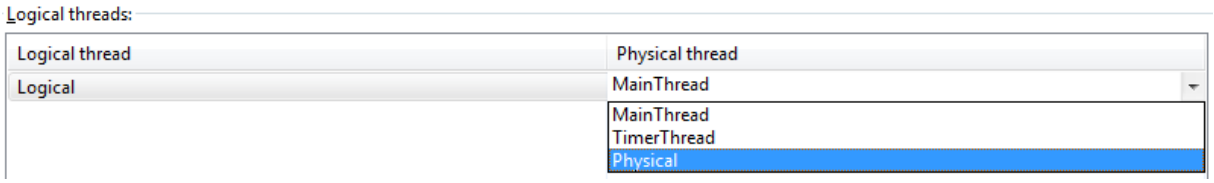
You may add and remove additional physical threads using the Add and Remove buttons.

For each physical thread you can set a few properties:

- **Name**
  The name of the thread.
- **Stack size**
  The stack size in bytes allocated by the thread. It is by default set to 20 kB. Thread stack sizes are set-up dynamically by code in the unit file (called “UnitName.cpp” by default). Note that some target platforms do not allow modifying the stack size of the main thread dynamically.

- **Priority**
  The priority of the thread. It is by default set to DEFAULT_MAIN_PRIORITY. See the file RTTarget.h in the RT services library for the available priorities.
- **Implementation class**
  The class in the RT services library that implements the thread. It is by default RTPeerController for all threads except the TimerThread which uses RTTimerController. You may specify your own thread implementation class instead to implement a custom controller, for example to use a different message handling strategy.

In the table “Logical threads” you can add logical threads with names that become available in the generated C++ code. These logical threads appear as a node under one of the physical threads in the “Physical threads” table. To map the logical thread to a different physical thread you can drag-and-drop it onto the desired physical thread. Alternatively you can use the drop-down menu in the "Logical threads" table:



Marking Models tab

A marking model is a platform specific model that may be used for transforming a platform-independent model to a specific target language. For example you may have a UML model which uses UAL (UML Action Language) as the action language. Such a model could be transformed to multiple target languages (C++, Java etc.), and the mapping to each such target language can then be controlled by a marking model.

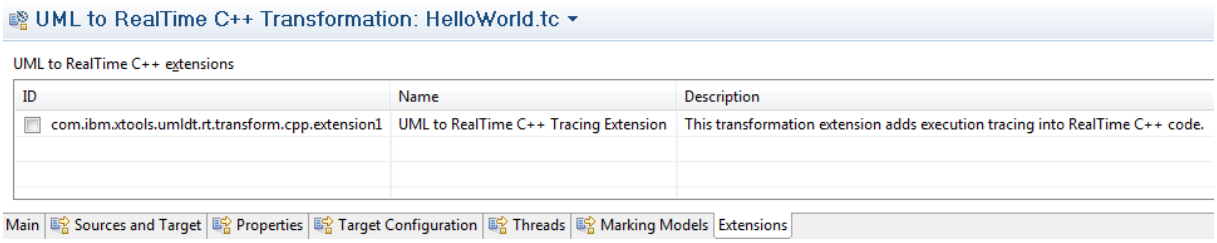
By default the C++ code generator does not consider any marking models, so most often the “Marking Models” tab will be empty. However, if you have developed an extension to the code generator that needs some data or settings to control how it should work, then instead of storing such information as stereotype tagged values directly in the model, it can be stored in a marking model. Thereby the same model can be transformed using different values for the settings, by using TCs which have different marking models specified.

Marking models are only supported by the classic builder, not by the model compiler.

Extensions tab

Generated C++ code can be customized by means of plugins that use the extension point `com.ibm.xtools.transform.core.transformationExtensions`. The Extensions tab will only appear if there is at least one such extension plugin available in the RSARTE environment. RSARTE does not ship with any extension plugins for the C++ code generator, so by default this tab will not be shown.

Here is an example of what the Extensions tab may look like when an extension plugin is available for customizing generated C++ code:



Use the checkbox for each extension to enable or disable it.

Code generator extensions are only supported by the classic builder, not by the model compiler.

Use of Variables in Transformation Configurations

As already mentioned above there are several variables that can be used in the values of TC properties. Most of these variables are defined in makefiles and the best way to learn about them is to look in the generated makefile and the makefiles that it includes. For example, many variables are defined in the file `<RSARTE INSTALLATION>/rsa_rt/C++/TargetRTS/libset/default.mk`.

However, there are also a few other special variables that can be used in a TC, and that will be expanded during the build of the TC. The table below lists those variables:

$(TOP_CAPSULE)	This variable can be used in the "Executable name" property of a C++ Executable TC. It expands to the name of the top capsule that is specified by the TC.
$(TCONFIG_NAME)	This variable expands to the name of the TC that owns the property where the variable is used. Use this variable in TC properties where you otherwise would hard-code the name of the TC.

User-defined Variables

It is possible to define your own variables and use them within a transformation configuration. This can be useful in order to create more generic transformation configurations which can be used in different environments. Rather than changing the TC in each environment, or using different TCs for different environments, the user-defined variables can be changed instead in order to accomplish the build variations that are necessary in a particular environment.

User-defined variables can be defined in different contexts. When a user-defined variable is used in a TC these contexts are scanned in a fixed order to locate a value for the variable. The following contexts are available (listed in the order in which they are scanned):

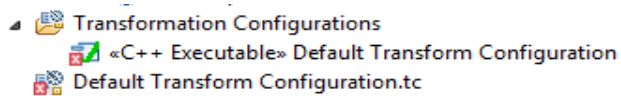
1. Workspace preferences defined in *Preferences – Run/Debug – String Substitution*
2. Environment variables defined in the system

User-defined variables can be used in the following places in a transformation configuration:

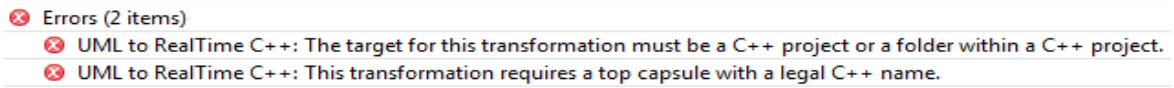
- The ”Location” property in the ”Sources and Target” tab. For example, a location specified as ”C:/users/$(USER)” can be used to let the location of the target project be dependent on a USER variable which each user of the TC can set-up differently.
- The ”Constants” property in a ”C++ External Library” TC. See External Constants for more information.

Creating Transformation Configurations

When you create a new model project using a template in the "UML Capsule Development" category, then you will get a default transformation configuration to use for building the model. Some templates use TCs in the traditional format while others use TCs in the new JavaScript format. Here is an example of a TC in the traditional format:

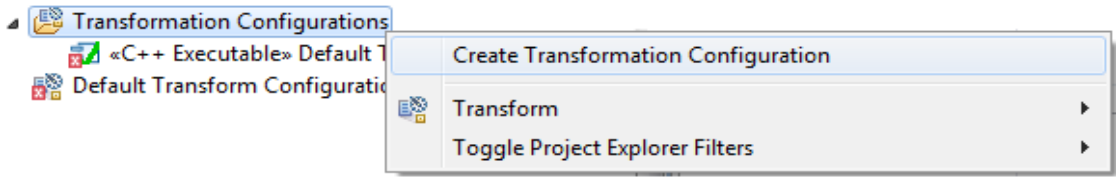


The error marker shows that the TC cannot be used until you have specified values for some mandatory properties. If you press the "Validate transformation configuration" button in the toolbar of the TC editor ( ) you will get a list of errors in the Problems view that tells you what you need to do before you can build the TC:



Here we can see that we need to specify the target CDT project for the TC (otherwise RSARTE doesn't know where to place generated C++ files), and that we must specify a top capsule (otherwise RSARTE doesn't know which capsule to incarnate at application start-up).

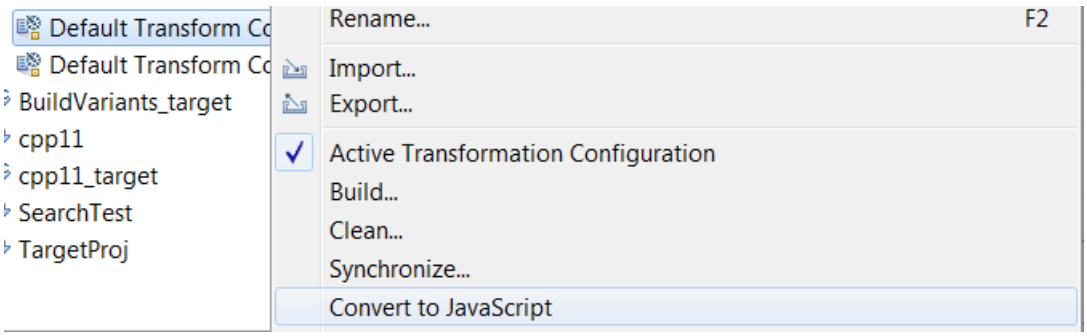
If you want to create additional TCs you may do so from the context menu of the "Transformation Configurations" virtual folder in the Project Explorer:



A wizard will appear that lets you specify the most important TC properties. Other properties can be set afterwards using the TC editor.

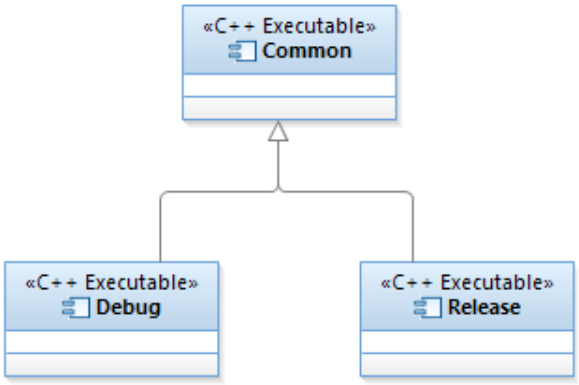
You can also create new transformation configurations by copy/paste of other transformation configurations in the Project Explorer.

TCs in the new JavaScript format can be created by converting TCs in the traditional format. Use the command "Convert to JavaScript" that is available in the context menu of the TC:

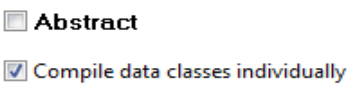


Inheritance and Abstract Transformation Configurations

You may create more than one TC for transforming a particular model. One reason for doing so could be to build different variants, for example a debug and a release version, of the same application. Often the majority of all properties in such variant TCs are identical, and there are only a few properties which have different values. To avoid duplicating properties in multiple TCs you can arrange your TCs in an inheritance hierarchy, where common properties are placed in an abstract TC which the other TCs inherit from. You cannot build an abstract TC. It only serves as a common place for properties that apply to all inherited TCs, unless they override them.
Here is an example of an abstract TC "Common" with properties that are shared by two inherited TCs "Debug" and "Release":

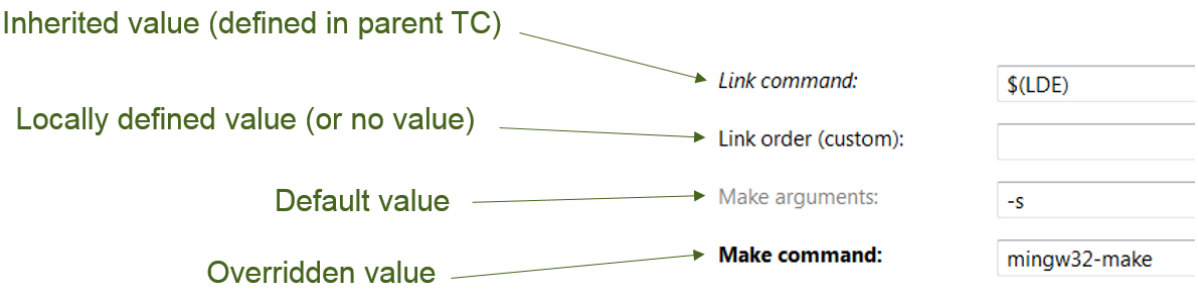


When you have set-up one or many inherited TCs the TC editor will mark the names of some properties in boldface. These are the properties which have been overridden with new values in the inherited TC. For example:



Here the “Abstract” property has been overridden. This means that the property Abstract == false is stored in this inherited TC. The same or a different value may be stored in the super TC.

There are also a few other visual indications of TC properties which are useful to know about. The picture below summarizes them:



There are several commands available in the context menu of the TC properties which help you work with inherited properties:

- **Revert to Saved**
  Restore the property to what is stored in the TC file. Useful in case you made some changes (not yet saved) that you want to undo.
- **Delete Value**
  Remove the overridden value for the property, so that the value stored in the super TC is shown instead. After this the property will appear in italics instead of boldface to show that it no longer is overridden.
- **Revert All to Inherited Values**
  Reverts (i.e. deletes) overridden values for all properties in the current TC editor tab.
- **Override**
  Overrides a property by storing the inherited value in the inherited TC file. This command is the opposite of "Revert to Inherited Value". After this the property will appear in boldface to indicate that it now is overridden.
- **Override All**
  Overrides all properties in the current TC editor tab.
- **Reinherit identical property overrides**
  Updates all TCs that inherit from this TC and overrides all properties in them that have identical values as stored in this TC.
- **Navigate to Inherited Value**
  Navigates to the inherited value for a TC property in a super TC. This command can be useful in order to find out from where a certain TC property gets its value (in case there are multiple super TCs).
- **Navigate to Parent Transformation Configuration**
  As above but works also for TC properties where the value is not inherited, but a default value defined in some parent (i.e. super) TC.

You can navigate to an inherited TC by double-clicking on an entry in the "Inherited transformation configurations" list. This, combined with the command "Show in Project Explorer" which is available in the toolbar menu of the TC editor, makes it easy to navigate between model projects that are related by means of TC inheritance.

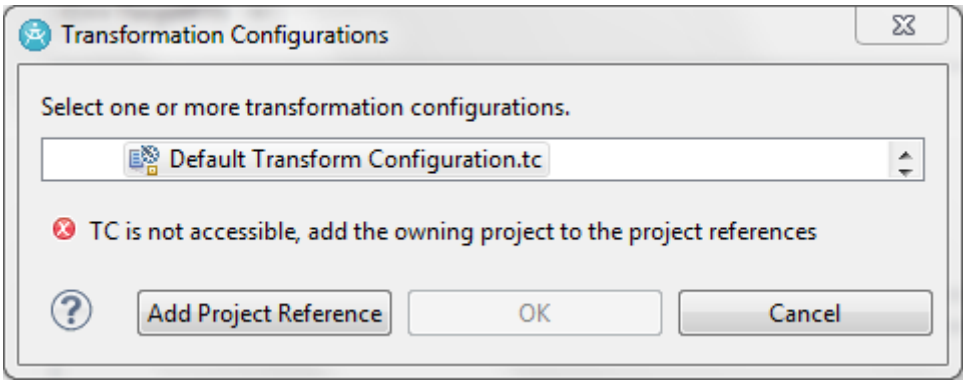
For further information about TC inheritance see this document.

Prerequisite Transformation Configurations

If a TC A has a TC B as its prerequisite it means that B has to be built first, before A can be built. You can use prerequisite relationships when you need to enforce a particular order in which different TCs are built. The typical example is to build an executable that links with a C++ library. The library must be built before the executable can be built, so we would use a prerequisite relationship from the TC that builds the executable to the TC that builds the library. Note that it is not possible to set-up a prerequisite relationship between two executable TCs – the prerequisite TC must be a library TC.

You can navigate to a prerequisite TC by double-clicking on an entry in the "Prerequisite transformation configurations" list. This, combined with the command "Show in Project Explorer" which is available in the toolbar menu of the TC editor, makes it easy to navigate between model projects that are related by means of TC prerequisite relationships.

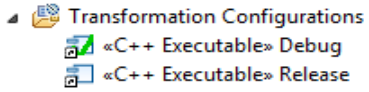
If you add a prerequisite relationship to a TC that is located in another project, you must set-up a project reference between your project and the other project before a prerequisite relationship can be added. If a project reference is missing an error will be shown in the dialog where you select the prerequisite TC.



You may press the “Add Project Reference” button to automatically set-up the required project reference.

Active Transformation Configurations

You can set a TC in your model project as active to tell RSARTE that you want to build this particular TC when building the project. To set a TC as active you can right-click the TC file or the component that represents the TC, and enable "Active Transformation Configuration" in the context menu. You can also change the active TC using the “Build Active Transformation Configuration” button as explained in Interactive Build. An active TC is marked with a green checkmark on the component representation of the TC under the "Transformation Configurations" folder in the Project Explorer. Here is an example:



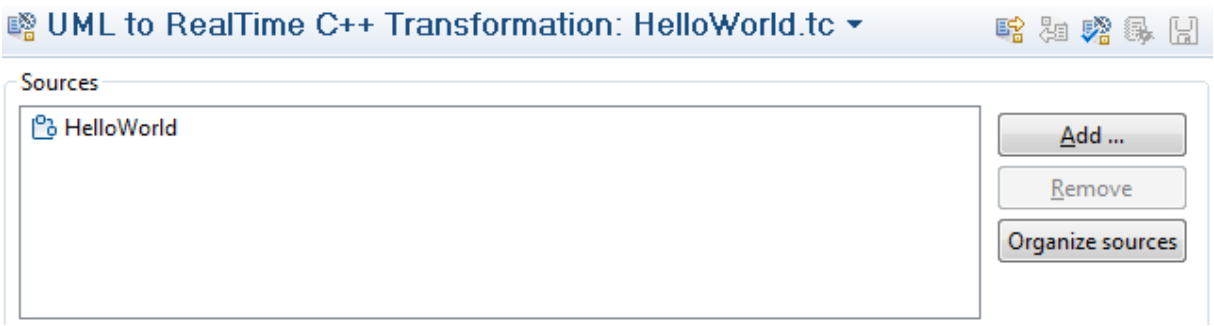
By default at most one TC can be active in each model project at any point in time. However, by enabling the preference *UML Development - Allow multiple active transformation configurations per project* you can set more than one TC to be active.

Managing the Sources of a Transformation Configuration

It is important to set-up the source elements of a TC properly. Ideally you want to transform only those elements that are really necessary to have in the C++ executable or library that results from building the TC. If you transform too many model elements the build will be slower since unnecessary C++ files have to be generated and compiled. Also, the produced binary files may become bigger than necessary.

To assist you with setting up the source elements in the best possible way, the TC editor has a wizard that offers several utilities that can help maintaining the Sources list. Note that this wizard is only available for TCs in the traditional format. If you use TCs in the new JavaScript format, you can instead use one of the other approaches that are described later in this chapter.

You start this wizard by pressing the "Organize sources" button that is located next to the Sources list in the TC editor. The Sources list needs to be non-empty, or a top capsule must have been specified, for this button to be enabled.



The wizard offers the following choices:

- **Optimize existing list**
  This choice performs an analysis of the current source elements. Elements that are not referenced by other elements in the list are suggested to be removed, while if there are elements that are indeed referenced, but not already in the list, these are suggested to be added. Depending on if the TC builds a library or an executable you may decide to follow the suggestions made by the wizard or not. For example, a library TC may include source elements that are not referenced locally, but are supposed to be used by elements built by other TCs. In that case you should not remove those elements from the Sources list.
- **Update the list by analyzing the top capsule**
  This choice is only available for C++ Executable TCs that has a specified top capsule. It will do the same analysis as described above, but instead of looking at elements in the Sources list, it will analyze the specified top capsule.
- **Replace packages with contained elements**
  This choice is available if there is at least one package in the Sources list. The wizard will suggest to replace those packages with the elements that they contain. This may be useful if you previously have transformed all elements in a package, but now want to be more

specific about what elements to transform. By first replacing packages with their contained elements, and then re-running the wizard to optimize the Sources list, you can obtain a more optimal list of source elements.

- **Replace contained elements with packages**
  This command does the reverse of the above, i.e. it suggests that all source elements that are contained by packages are to be replaced by those packages. This may be useful if you realize that there are packages in the model that do not contain any elements that need not be transformed. In that case you may prefer to just list the packages as source elements, and not all the elements they contain.

Note that in all cases above the analysis is only done using references that are available in the model. C++ code snippets are not considered. That is another reason why the wizard requires your review and confirmation before the Sources list actually gets updated. If you disagree with the suggestions made by the wizard you can just cancel it to leave the Sources list unmodified.

If you have many TCs to maintain it may not be practical to open each of them in the TC editor to update them using the Organize Sources wizard. Especially not if this is something you need to do frequently. There are two approaches to address this problem:

- Use an Ant task for automating what the Organize Sources wizard does.
- Avoid specifying a list of source elements completely, by using the “Detect Source Dependencies Automatically” feature.

The chapters below describe these possibilities.

Updating Sources Using Ant Task

You may write an Ant build file and use it for analyzing and updating the Sources list of several TCs in one go. The Ant task you use for this is `com.ibm.xtools.umldt.organizeTransformationConfiguration`. This Ant task can both perform the analysis of TCs, as well as to actually update the TCs based on the results of that analysis. The Ant task also supports to save the results of the analysis in an XML file, which later can be read by the Ant task to perform the actual updates. This is useful if your TC files are read-only, for example because they are in a CM system such as ClearCase. You can then write a script which first runs the wanted analysis on TCs and saves the result in XML files. Then you can make writable those TCs that the analysis suggested to be updated. And finally you can use the XML files to perform the actual updates of the TCs.

The following attributes are supported by the `organizeTransformationConfiguration` Ant task:

- **rule**
  Specifies the kind of analysis to run on the TC. This attribute is optional, and is not used when you use the Ant task for updating a TC based on a previously made analysis. The following values can be used for this attribute:
  - `AddMissingDependencies`
    This choice runs the same analysis as when choosing "Optimize existing list" in the Organize Sources wizard.

- `DetectCapsuleDependencies`
  This choice runs the same analysis as when choosing "Update the list by analyzing the top capsule" in the Organize Sources wizard.
- `ReplaceContainersByContainedElements`
  This choice runs the same analysis as when choosing "Replace packages with contained elements" in the Organize Sources wizard.
- `ReplaceElementsByContainers`
  This choice runs the same analysis as when choosing "Replace contained elements with packages" in the Organize Sources wizard.

- **outputFile**
  Specifies an XML file where the results of the chosen analysis will be stored. This attribute is optional, and if you don't use it the TC will be updated directly.
- **inputFile**
  Specifies a previously generated XML file containing the results of analyzing a TC. The Sources list of the TC will be updated according to the contents of the XML file. This attribute is optional and you don't use it when you only want to analyze the TC.
- **transformConfig**
  Specifies one or many TCs to analyze and/or update. You may specify more than one TC by using wildcards such as * or ?. Paths should be relative to the workspace. For example, the path "/MyProj/*.tc" will analyze and/or update all TCs in the project called MyProj. This attribute is optional if you specify an input file. In that case the TCs to update are given by the contents of the input file.

Here is an example of an Ant build file which uses the organizeTransformationConfiguration Ant task:

```
<project name="OrganizeSources" default="analyze">
<target name="analyze">
    <com.ibm.xtools.umldt.organizeTransformationConfiguration
     rule="DetectCapsuleDependencies"
     outputFile="C:/temp/out.log"
     transformConfig="/OrganizeSources/*.tc"
    />
</target>
<target name="update">
    <com.ibm.xtools.umldt.organizeTransformationConfiguration
     inputFile="C:/temp/out.log"
     transformConfig="/OrganizeSources/*.tc"
    />
</target>
</project>
```

The target "analyze" performs analysis of TCs and writes results to an XML file. The target "update" uses the written XML files to update the TCs with the results from the analysis.

Note that the Ant task for updating the Sources list of a TC is only supported for TCs in the traditional format.

Detect Source Dependencies Automatically

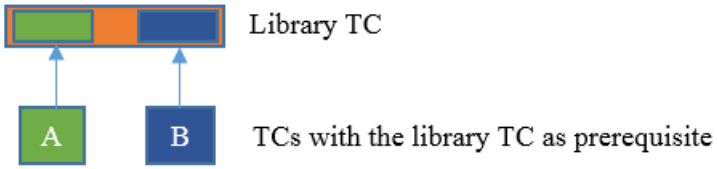
A different approach to automating the analysis of which source elements that need to be transformed is to use the preference *UML Development – Real Time C++ Transformation –*

Detect source dependencies automatically. When this preference is enabled the Sources list of a TC can be left empty, or only contain a few elements. When the TC is built the C++ transform will include dependent source elements automatically, without actually changing the TC itself. For an executable TC the top capsule is automatically added as a source element, while for a library TC you need to specify at least one element in the Sources list which the analysis can start from.

It's important that all users who build a model have set the "Detect source dependencies automatically" preference consistently. When it is used you can no longer look at the Sources list of a TC to understand which elements that will be transformed when the TC is built. Also note that when this preference is set, the prerequisites of the TC are no longer used when determining which elements that need to be transformed. It's solely the references in the model that determines which elements that need to be included in the transformation. Therefore it's very important that references in the model are correctly set-up, so they are bound to the expected model elements. If you find that unexpected model elements get transformed when using this feature, you can turn on the preference *UML Development – Real Time C++ Transformation – Report details about automatically added source elements*. Then you will get information messages about which elements that are considered necessary to transform, and why.

Context Sensitive Library Builds

It is common that the source elements of a TC with prerequisite libraries only reference a small subset of all elements that are available in those libraries. This is because a library is typically designed for being used in many different contexts (executables and/or other libraries) and in each such context only some limited part of the library is used. For example, consider the situation when a library TC is the prerequisite of two different TCs A and B:



The source elements of A only reference some parts of the library (marked in green above), while the source elements of B reference a different subset of the library (marked in blue above).

By default a library TC is built into a library which contains all its source elements. All executables and libraries which have that library as a prerequisite can then link with it. However, if the library is big, and you only want to build one particular TC that uses it, the time it takes to transform and compile source elements that are not referenced is actually wasted.

To address this problem, RSARTE supports a feature called "Context sensitive library builds". It is controlled by a preference *UML Development – Real Time C++ Transformation – Context sensitive library builds*. Just like the "Detect source dependencies automatically" feature it is based on analyzing which elements that are actually referenced by the source elements of the TC that is built. All elements from prerequisite TCs that are referenced will be transformed and built into object files. These object files are then put together into a library which is linked with the executable or library that corresponds to the top-level TC that is built.

This means that the result of building a library TC will be different depending on how it is built:

- If it is built directly (i.e. as a top-level TC) then all specified source elements will be built into the library as usual.
- If it is built indirectly (i.e. as a prerequisite of another TC) then only those source elements of the library that are actually used by source elements of that other TC will be built. The result of the build will be a limited version of the library that only contains object files for each source element that is used. This version of the library can hence not be reused by other TCs that are built, and it is therefore placed in the output folder of the top-level TC.

Note that the “context sensitive library builds” feature is only supported when using inclusive makefiles. If you attempt to use it with recursive makefiles the build will be aborted with an error.

If you only want to build some TCs in a context sensitive way, you can set a TC property called “Context sensitive library build”. The property applies for the TC itself and also for all its prerequisite libraries.

Automated Source Management and External Code

The two strategies for automating the management of TC source elements described in Detect Source Dependencies Automatically and Context Sensitive Library Builds both rely on correct references in the model, since that is what is used for determining what gets built. If you have external C++ code that references model elements, you must therefore make sure that these references are formally represented as dependencies in the model. Otherwise the build may skip those referenced elements, which will lead to build errors. The recommended way to include external code that references model elements is to use artifacts, since you then can use regular dependencies on the artifact to represent the references to other elements that exist in the code. However, if you have external code outside of the model that contains references to code generated from the model, then you can follow the steps outlined below:

1. Create a special class in the model that contains the elements that are referenced from external C++ code. Open the Properties view and go to the ”C++ General” section. Set ”Generate files” to Header so that only a header file gets generated for this class.

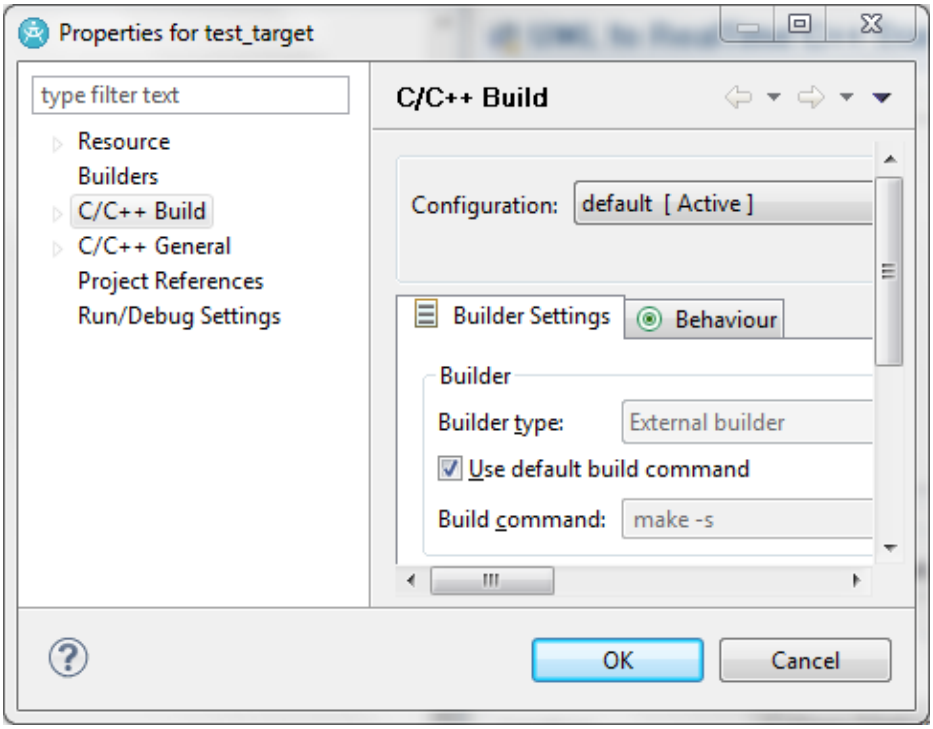


2. For this special class add dependencies to all UML model elements that are referenced by external code.
3. Finally create dependencies to this special class from all elements of the model that are referenced from other models. This ensures that whenever you build those other models, the model elements that are referenced from the external code will also be included into the build.

Building Generated Code

Once the model has been transformed to C++ code in a CDT project, the next step in the build process is to build the generated code (i.e. to compile and link it). This is done by building the

CDT project. Depending on the TC property "Automatically create and update target project" the CDT project will be configured to be built in different ways. If this property is unset the CDT project is assumed to have been properly set-up so it can be built directly. However, if the property is set the C++ transform will also generate a makefile which the CDT project will build using a make tool (acting as an external builder from CDT's point of view). You can open the Properties dialog on a CDT project if you are uncertain how it will be built. Here is an example of a CDT project that will be built using a make file:



To learn more about how a CDT project is built you may refer to the Eclipse CDT documentation. In the rest of this chapter we will look at the case when the build takes place using a generated makefile.

A big advantage with building a generated C++ project using a makefile is that the workspace will only be locked for modifications during the time it takes to run the C++ transformation. Once all code has been generated (including CDT projects and makefiles), the rest of the build is done by make on the makefiles and during this time the workspace is not locked. This means that you can continue to work in the model as soon as the transformation phase is completed. Usually this phase is much shorter than the time it takes to run make on the makefile.

If you use the model compiler as your build tool, this can be further improved so that the workspace only is locked when generating the makefile. This makefile can then drive the complete build, including C++ transformations. However, it is also possible to use the model compiler only for building generated code. See Two Types of Builds – Model Compiler versus Classic Builder for more information about the model compiler.

If you are using the classic builder, you should not attempt to modify the model or its code snippets while the C++ transformation is ongoing. Since the workspace is locked during this

time no changes can be committed to the model. The changes may therefore easily be lost if the editors revert to the saved model, or if the C++ code generator overwrites a C++ file that you have modified. Once the message "Transformation completed successfully" is printed in the UML Development console, the transformation phase is completed and you may again edit the model and its code snippets.

Makefile Generation

There are two kinds of makefiles that may be generated by the C++ transform (controlled by the preference *UML Development - Batch Make Type*):

- **Recursive makefiles**
  In this case the makefile that is generated for the TC that is built will recursively invoke make on makefiles that are generated for the prerequisite TCs. There will hence be one invocation of make for each TC that is part of the build.
- **Inclusive makefiles**
  In this case the makefile that is generated for the TC that is built will include all rules that are necessary to build both the TC itself and all its prerequisite TCs (both direct and indirect). There will hence only be a single invocation of make.

The default is to generate recursive makefiles, since it yields makefiles that are shorter and easier to read. However, the performance of the build may be improved if you instead choose to generate inclusive makefiles. This is in particular true for make tools that support parallel builds, since the distribution of build tasks on different computers often can be done more efficiently with a single makefile as input. Note that the model compiler only supports inclusive makefiles.

The makefile that is generated for the TC that is built is called `batch.mk`. Note that in spite of its name this makefile is used both for interactive builds and batch builds. The build is done by running the following make command:

```
<make-command> -s -f batch.mk all
```

where `<make-command>` is specified in the "Target Configuration" tab in the TC.

In addition to `batch.mk` a makefile called `Makefile` is also generated. This makefile (and included makefile fragments 0.mk, 1.mk etc.) is invoked from `batch.mk`.

For historical reasons (legacy from RoseRT) the makefiles generated by the classic builder precede some make commands with the character '@' in order to suppress the command name in output messages that are printed. You can use the preference *UML Development – Real Time C++ Transformations – Do not generate '@' for commands* to specify make commands that should not be preceded by the '@' character. Use a comma-separated list, for example

```
$(FEEDBACK), $(IN)
```

Two Types of Builds – Model Compiler versus Classic Builder

RSARTE contains two different tools for building a model:

- The classic builder
- The model compiler

The preference *UML Development – Real Time C++ Transformations – Build type* controls which build tool to use.

The main difference between these build tools is how C++ transformations take place. With the classic builder this happens inside RSARTE, which means that the workspace will be locked for modifications until this phase is completed. With the model compiler C++ transformations are performed by a separate application (the model compiler) outside of RSARTE. This means that you can continue to work in RSARTE right after the makefile has been generated. However, even if this means you can start to edit the model while the model compiler runs, it is not recommended to save your changes until the build is completed, since otherwise it may be hard to know exactly which changes that went into the build.

Contrary to the classic builder, the model compiler supports to drive the entire build process from a makefile. If the preference *UML Development – Real Time C++ Transformations – Include make rules for model-to-code transformations* is set the generated makefile will contain rules not only for building generated code, but also for generating the code from the model. Having such complete makefiles can improve build performance on systems that support parallel execution of make rules. However, for small models, the overhead of launching the model compiler from the makefile can outweigh the performance benefits. You therefore need to measure whether or not it is beneficial in your particular environment to use this preference. If you decide to use it you can set the radio button below it to “Separate”, “Joint” or “Full” to control how many times the model compiler will be launched when processing the makefile. Once again it’s not possible to give a general recommendation on which of these options that will give the best build performance in your particular environment. You need to test the different options to see what gives the shortest build time for your model and environment.

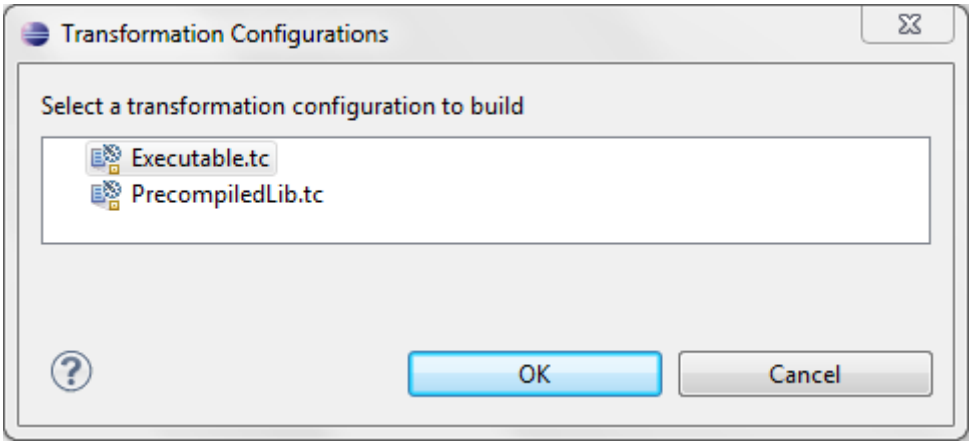
Interactive Build

An interactive build (sometimes also called IDE build, or GUI build) is triggered by pressing the “Build Active Transformation Configuration” toolbar button:



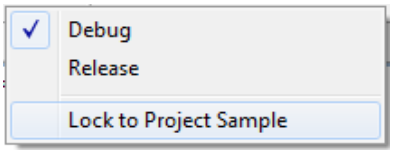
This button is always enabled and builds the active TC in the selected project. If a project is not directly selected in the Project Explorer, RSARTE uses the project to which the selected element belongs. For example, if you select a capsule the Build Active TC button will build the active TC in the project that contains the capsule. If the project contains multiple active TCs they will all be built, one by one starting with the first one.

If the project contains TCs but none of them is marked as active, then a dialog will appear which lets you choose which of the project's TCs to activate and build:



If the project does not contain any TCs at all, the dialog will let you choose a TC from another project in the workspace.

Often you always want to build the same TC no matter what element that happens to be selected at a particular point in time. In this case you should lock the Build Active TC button to a particular project. This is done from the menu that appears when clicking on the black triangle to the right of the button:



Once you have locked the button to a project, it is no longer sensitive to the selection and you can press it to build the active TC of that project at any time.

Using this button menu you can also choose to build another TC that is not active. When you choose the other TC it will be marked as active. After that you can press the Build Active TC button to build it.

Important: Eclipse has an option to build projects automatically (”Build Automatically” in the Project menu). Ensure that this option is off when building a TC from the user interface. Otherwise the generated CDT project may not get built when building a TC, and the build will terminate after the transformation phase.

Build Messages

As soon as the build of the TC has been started, RSARTE will pop-up the UML Development console in the Console view. This is where all build messages will be printed. You may want to pin this console to ensure that it stays open during the entire build process.

Build messages that are printed to the UML Development console includes:

- TC validation messages (same as are reported when running an explicit validation of the TC)

- Transformation messages (e.g. warnings or errors detected during C++ code generation)
- Compiler messages
- Linker messages

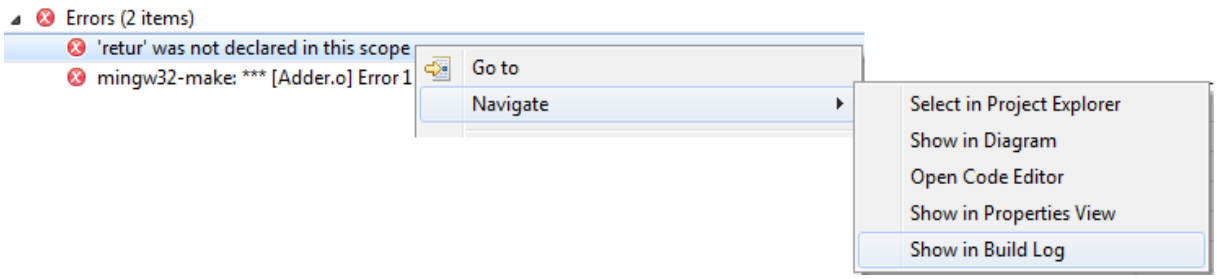
You can double-click on messages in the UML Development console in order to navigate to the location of the message. This could either be a TC, an element in the model, or it could be somewhere in the generated C++ code.

Note that when the model compiler is used, compiler and linker messages are not printed directly to the UML Development console, but to a build log. A link to the build log file is printed in the UML Development console together with a message if the compilation failed due to errors. For example:

```
08:28:44 : INFO : Build log file is created
08:28:47 : ERROR : Compilation failed
```

You can click on the Build log link to open the build log in RSARTE.

Most messages also have a representation in the Problems view, and when a message in the UML Development console (or in the Build log) is double-clicked, the corresponding problem in the Problems view will be selected. From its context menu you can choose to navigate to other locations where the problem can be fixed, or where to find more information about it. You can also choose to navigate to the build log to see the context of the problem. Sometimes this is necessary in order to understand how to fix it.



You can manually remove problems from the Problems view by deleting them. For example, you may want to delete problems that you have looked at or fixed. When you start a new build you may want to automatically delete all or some of the problems that already exist in the Problems view so that it becomes easier to see the problems that are caused by that particular build. To control what to do with the Problems view when you start a new build you should set the preference *UML Development - Clean problems view before build*:

- **Do not clear problem markers**
  Use this if you prefer to manually delete problems from the Problems view.
- **Clean problem markers for all built projects**
  This choice will delete those problems that are reported on the projects that are built. This includes both the source model projects (for example warnings on model elements produced by the code generator) and the target CDT projects (for example compilation errors). Problems reported on projects that are unrelated to the build will remain. The idea here is that only those problems that were caused by building the same TC previously

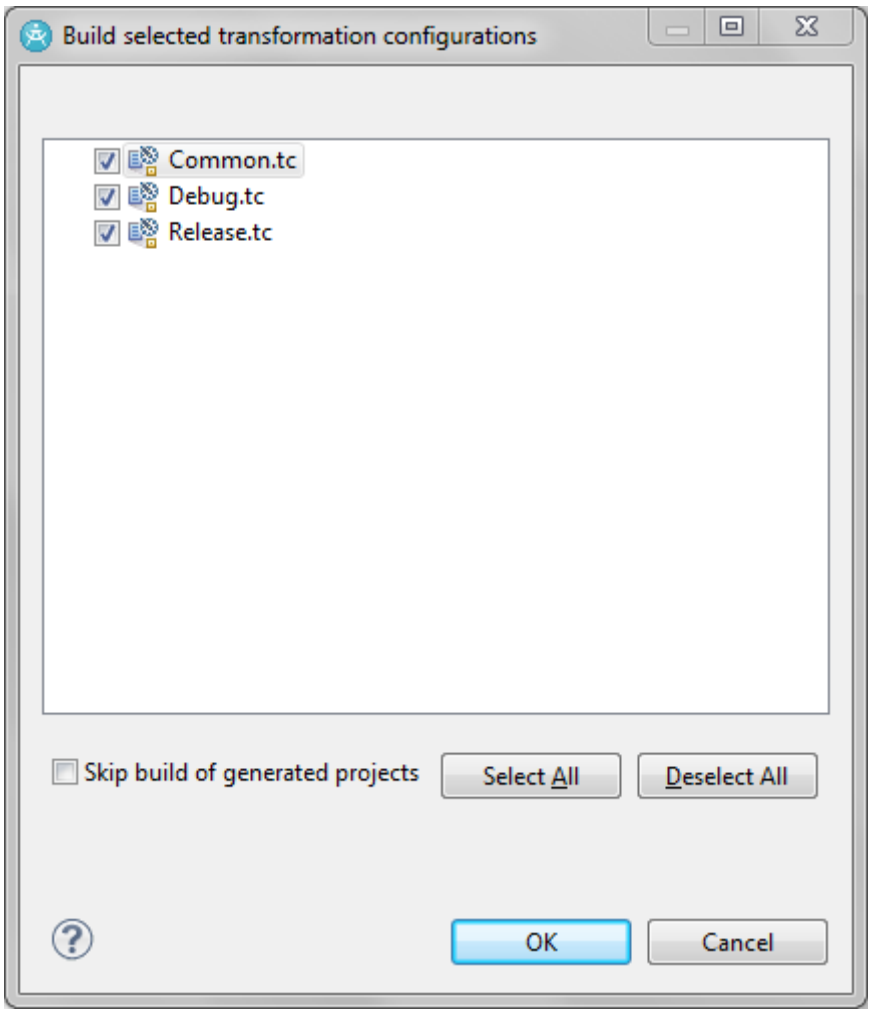
should be removed, before the TC is built again. Any problems that remain will then re-appear as a result of the new build.

- **Clean problem markers for all built project and all CDT projects**
  This choice works the same as the above, but in addition problems that are reported on other CDT projects are also removed. This may be useful if you have external C++ code that gets included in the build, for example using C++ External Library TCs (as described in External Libraries).
- **Clean all problem markers**
  Use this if you want to always remove all old problems when starting a new build.

Building Multiple Transformation Configurations

An alternative to building a TC using the ”Build Active Transformation Configuration” button is to invoke the command "Build..." in its Project Explorer context menu. This command also works when you have multiple TCs selected, and is therefore in particular useful if you want to build several unrelated TCs.

The "Build..." command brings up a dialog where you can select the TCs that you want to build (by default those that were selected in the Project Explorer):



When you press OK all the marked TCs will be built one by one.

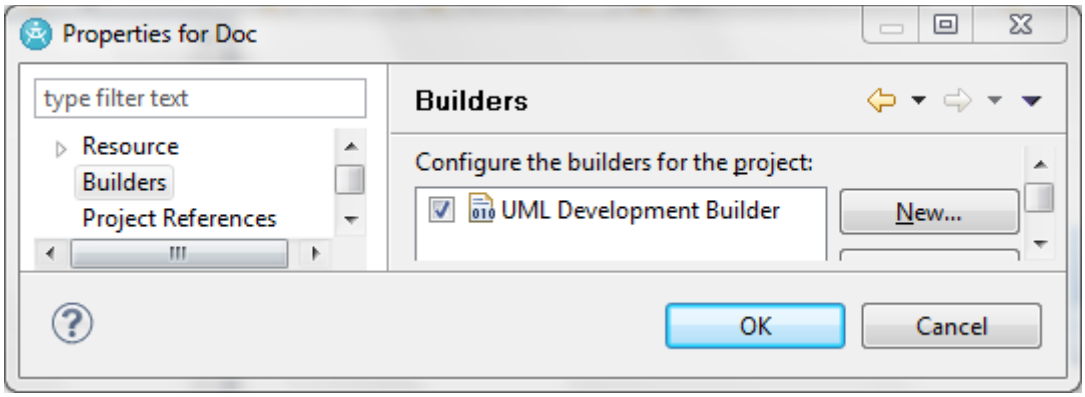
If you only want to run the transformation step for the TCs (i.e. only generate the code but not build it) then you can mark the checkbox "Skip build of generated projects".

Note that currently the model compiler does not support building multiple transformation configurations at the same time using the "Build..." context menu command.

Alternative Ways to Trigger an Interactive Build

Let's go into some more detail about what happens when performing an interactive build, and some alternative ways to trigger it.

An interactive build is controlled by an Eclipse builder called the UML Development Builder. If you look in the properties of an RSARTE model project you will see in the Builders tab that the project is built using this builder.



The Eclipse builder framework provides user interface that allows you to invoke the UML Development Builder on a project. For example, it provides a "Build Project" command in the context menu of a project (available if *Project – Build Automatically* has been turned off). When the UML Development Builder is asked to build an RSARTE model project it will look for active TCs in that project. All TCs that are marked as active will be built, one by one.

Usually there should be no need to perform interactive builds through the general Eclipse builder user interface. Most often there is one particular TC in the project that shall be built, and therefore it is better to use the “Build Active Transformation Configuration” button for building that specific TC as described previously. If the project contains multiple TCs that should be built, the context menu command "Build..." can be used.

A special feature of an Eclipse builder is the ability to automatically build a project as soon as it or its content changes. This behavior is controlled by the preference *Project – Build Automatically*. This feature is not appropriate to use with the UML Development Builder, especially not when a makefile is used for building generated C++ code. It is therefore strongly recommended to have this preference turned off when building RSARTE C++ models (it is turned off by default in RSARTE).

Batch Build

A batch build (sometimes also called headless build, or non-GUI build) can be done in two ways. The recommended approach is to use the model compiler, since that avoids the need to run a headless instance of RSARTE (which both takes time to launch, and also requires the DISPLAY variable to be set on Unix). It is also possible to perform batch builds by running a headless instance of RSARTE.

Model Compiler

The model compiler is a stand-alone command line tool which does not have any dependency to Eclipse or something that requires a user interface. Therefore it is possible to run the model compiler also without setting the DISPLAY variable on Unix.

To perform a batch build using the model compiler you can either call it directly from the command line, or (which is more typical) from a script. There is a separate documentation for the model compiler which covers all its command line options. You can find this document in the built-in Help of RSARTE at *RSARTE User's Guide – Wiki – Building – Model Compiler*.

Headless RSARTE

You can perform a batch build by running a headless (i.e. non-GUI) instance of RSARTE. The command line argument `-application` is used to specify an Eclipse application which triggers the build. There are Eclipse applications available which run a TC, build a CDT project etc. There is also an Eclipse application which runs an Ant build file, which in turn may invoke Ant tasks that trigger the build.

To launch a batch build invoke the following command from the command-line:

```
<RSARTE> -nosplash -application <APPLICATION> -data <WORKSPACE> <APPLICA-
TION ARGS>
```

<RSARTE> is the path to the “eclipse” application in the RSARTE installation directory.
<APPLICATION> is the Eclipse application to use for triggering the build. Available applications are described below.
<WORKSPACE> is the workspace to use (a directory).
<APPLICATION ARGS> is a list of arguments that depends on which application that is used.

The `-nosplash` flag ensures that the Eclipse splash screen does not appear when launching the batch build.

Usually you want build messages to be printed to the command line console (`stderr` or `stdout`) when running a batch build. To achieve this also for build messages that are routed through CDT (such as compilation errors), you should add the following command line argument:

```
-vmargs -Dorg.eclipse.cdt.core.console=org.eclipse.cdt.core.systemConsole
```

This argument tells CDT to use the command-line (system) console instead of the regular UI console (called C-Build).

Building with an Ant Build File

To run an Ant build file use the `org.eclipse.ant.ui.antRunner` application[2]. The command-line looks like this:

```
<RSARTE> -nosplash -application org.eclipse.ant.ui.antRunner -data <WORK-
SPACE> -buildfile <ANT BUILD FILE> <ANT TARGET>
```

<ANT BUILD FILE> is the Ant build file to run.
<ANT TARGET> is the name of a target within the Ant build file to run.

You can use the Ant support in Eclipse to write the Ant build file to use. Refer to the Eclipse documentation for more about writing and running Ant build files from within Eclipse.

RSARTE provides a large number of Ant tasks that you can use from your Ant build file. Most of these Ant tasks are built-in Ant tasks and tasks that are provided by Eclipse. The latter give access to useful functionality from Eclipse plugins, such as building a project, refreshing a resource etc. There are also a few Ant tasks that are specific to RSARTE and which provide the possibility to build TCs and target CDT projects. They are described below. Note that you can also create your own Ant tasks to be used from your Ant build file. Refer to the Eclipse documentation for more information on how to do this.

There are two Ant tasks provided by RSARTE which give you the possibility to build TCs and target CDT projects.

com.ibm.xtools.umldt.buildTransformationConfiguration
This task builds a specified TC. The following attributes are supported:

- **transformConfig**
  The path of the TC file to build. This is a workspace relative path. This attribute is required.
- **invokeTargetBuild**
  This is an optional attribute that specifies if the CDT project(s) that are generated when building the TC should also be built afterwards. You can set this attribute to “incremental”, “full”, “clean” or “cleanBinaries”. Incremental build means that only modified files will be built. In a full build everything is rebuilt from scratch. The value “clean” means that the TC will be cleaned, i.e. all generated files (and the CDT project) will be removed. If “cleanBinaries” is used the TC will be cleaned by cleaning its CDT project (done by calling `make clean` on the makefile that is generated from the CDT project). This means that all generated source code and the CDT project are kept, but the built binaries will be removed.

  If you do not give this attribute a value (or give it a value other than those mentioned above) then the generated CDT project(s) will not be built after the TC has been run.
- **markerFailureThreshold**
  This optional attribute only has an effect if “invokeTargetBuild” is set to “incremental” or “full”. In that case the Ant task will check after each built CDT project that there are no

2 Eclipse also provides an Ant runner without UI dependencies: `org.eclipse.ant.core.antRunner`. However, this Ant runner cannot be used for building RSARTE models in batch. If you need to perform batch builds where no UI is available, you need to use the model compiler instead.

messages (a.k.a markers) with a severity that is higher or equal to the value of this attribute. You can set this attribute to “info”, “warning” or “error”. Any other value is ignored. The Ant task will fail if a message with an equal or higher severity than the specified threshold occurs. For example, set this attribute to "error" to abort the build if an error occurs.

- **batchMakeType**
  This optional attribute can be set to “inclusive” or “recursive”. It determines what kind of makefile to generate for building generated C++ files. By default the makefile will be recursive. See Makefile generation for more information.

com.ibm.xtools.umldt.buildTransformationTargetProjects
This task builds CDT project(s) that have previously been generated when building a TC. The following attributes are supported:

- **transformConfig**
  The path of the TC file which generated the CDT project(s) to build. This is a workspace relative path. This attribute is required.
- **buildKind**
  This required attribute specifies how to build the target CDT project(s). You can set this attribute to “incremental”, “full” or “clean”. Incremental build means that only modified files will be built. In a full build everything is rebuilt from scratch. The value “clean” means that the target project(s) will be cleaned, i.e. all files that were created when building the project previously (for example binaries) will be removed.
- **markerFailureThreshold**
  This optional attribute only has an effect if “buildKind” is set to “incremental” or “full”. In that case the Ant task will check after each built CDT project that there are no messages (a.k.a markers) with a severity that is higher or equal to the value of this attribute. You can set this attribute to “info”, “warning” or “error”. Any other value is ignored. The Ant task will fail if a message with an equal or higher severity than the specified threshold occurs. For example, set this attribute to "error" to abort the build if an error occurs.

Here is an example of an Ant build file which uses the RSARTE Ant tasks.

```
<?xml version = "1.0"?>
<project name = "build" default = "generate-and-compile">

    <!-- Build a TC and then build generated target CDT projects. -->
    <target name = "generate-and-compile">
        <com.ibm.xtools.umldt.buildTransformationConfiguration
            invokeTargetBuild      = "full"
            markerFailureThreshold = "error"
            transformConfig        = "myProj/MyTC.tc"/>
        <echo message = "Build completed." level = "info"/>
</target>

    <!-- Build the target CDT projects of a TC.
         Note that this assumes they were previously generated. -->
    <target name = "compile">
        <com.ibm.xtools.umldt.buildTransformationTargetProjects
            buildKind              = "incremental"
            markerFailureThreshold = "error"
            transformConfig        = "myProj/MyTC.tc"/>
```

```
        <echo message = "Build completed." level = "info"/>
    </target>

</project>
```

To integrate the building of RSARTE models into a bigger build system, you may use a make file which invokes your Ant build file. Here is an example of a makefile (named MyMakefile) which uses the above Ant build file (named build.xml):

```
WORKSPACE = C:\\Users\\Mattias\\workspaces\\batch_build_workspace
VMARGS    = -vmargs -
Dorg.eclipse.cdt.core.console=org.eclipse.cdt.core.systemConsole

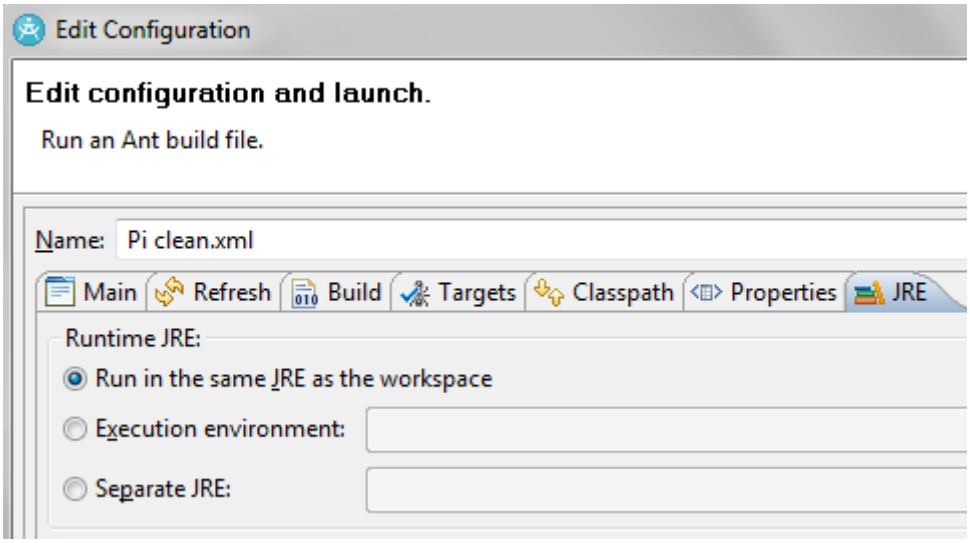
default : generate-and-compile
      echo $@

compile generate-and-compile :
      echo $@
      C:\\RSARTE\\eclipse \
            -nosplash \
            -application org.eclipse.ant.ui.antRunner \
            -data $(WORKSPACE) \
            -buildfile build.xml \
            $@ \
            $(VMARGS)
```

Now we can invoke this makefile in order to perform the batch build:

```
make -f MyMakefile
```

Note that you can also run an Ant build file within the RSARTE IDE (useful for example when writing or testing it). To do this simply right-click in the Ant file editor and choose *Run As – Ant Build*. You should have set-up the launching of the Ant file so that it gets run in the same Java Runtime Environment (JRE) as is used by the RSARTE IDE. This can be done from the JRE tab in the Edit Configuration dialog that appears if you invoke *Run As – Ant Build...* from the file editor.



Building Directly from the Command-Line

If you don't need the power of Ant to perform your batch builds you can trigger a build directly from the command-line by using the following Eclipse applications:

- `com.ibm.xtools.transform.core.transformationRunner`
  This application runs one or many TCs. It has an argument `-transformConfigs` which lets you specify a comma-separated list of TC files to run. Each TC file is specified by means of its workspace relative path. Note that running a TC is not the same as building it. When running a TC the target CDT project and all files will be generated, but they will not be built.
- `org.eclipse.cdt.managedbuilder.core.headlessbuild`
  This application builds a CDT project. See the CDT documentation for details about the arguments that are supported by this application.

Here is an example of running a transformation directly from the command-line:

```
C:\eclipse-cpp-oxygen\eclipse\eclipse.exe -data C:\Users\Mattias\workspace
-nosplash -application com.ibm.xtools.transform.core.transformationRunner -
transformConfigs "/MyProject/MyTC.tc"
```

Build Variants

We have already mentioned above the need to build different variants of an application. There could be many reasons why this is needed. For example:

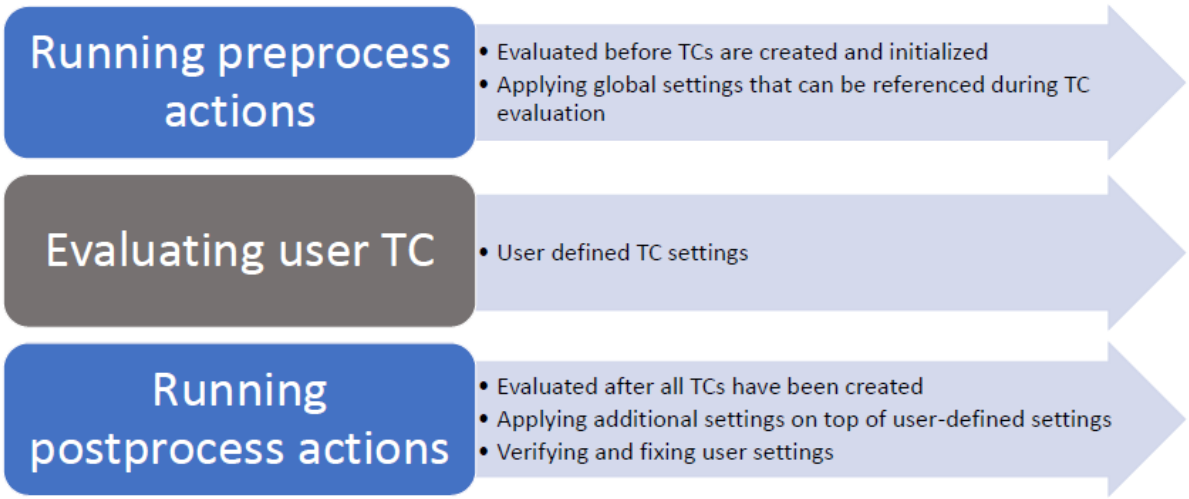
- Create a debug versus a release build of the application
- Add special instrumentation to the application in order to detect run-time errors
- Build the application for different target platforms
- etc.

TC inheritance can be one way to structure TCs to make it possible to build different application variants while reducing duplication of TC properties. Use of variables in TC properties is another option. However, none of these mechanisms have proven sufficient for large models with many build variants. With an increasing number of ways to parametrize a build, the number of possible build variants grows very quickly. Defining a new build variant may require a large number of TCs to either be created or updated. This soon becomes tedious, and is also a usability problem for users that have a huge number of TCs to choose from when deciding what to build.

The model compiler provides a solution for this problem that allows you to only have one set of TCs that are common for all variants of an application that need to be built. The idea is to only store properties that are common for all build variants in the TCs, and dynamically add or modify the TC properties that are specific for a particular build variant. In essence this makes it possible to create the variants of a TC dynamically at build-time without having to manually create a TC file for each and every build variant. Note that the model compiler only supports build variants for TCs in the new JavaScript format.

The dynamic manipulation of TC properties is achieved by means of writing one or many scripts (using JavaScript) that are run by the model compiler. Such scripts can be run either just before ("pre-processing") or just after ("post-processing") the default interpretation of TC

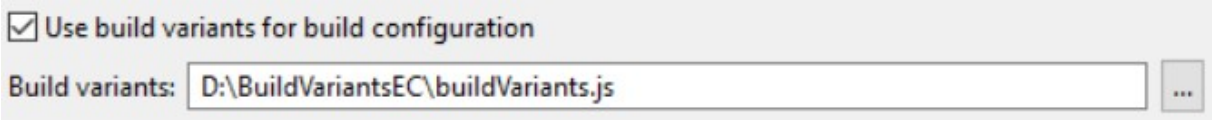
properties. Depending on the kind of build variant you want to implement you can choose to write "pre-processing" or "post-processing" scripts (or both). In a pre-processing script you can for example assign values to variables that are referenced in TC properties, while in a post-processing script you can directly modify the properties in the built TC or its prerequisites. All modifications performed by the scripts are transient, which means that they will never be stored in a TC file, but will only be used in the current build.



The model compiler provides a Transformation Configuration Framework (TCF) which is a JavaScript API for working with TCs. It provides functions for reading and writing TC properties, traversing prerequisite TCs, working with TC inheritance and much more. Together with the built-in functionality of JavaScript this provides for a very flexible and powerful way of dynamically manipulating TCs in order to build desired application variants.

There is also another JavaScript API, known as the Build Variant Framework (BVF), which allows to define which build variants to make available for users when they build a TC (either from the user interface or command-line). The idea here is that an advanced user (a build expert) writes the scripts that implement the different build variants, and also defines the user interface with the controls other users will see when they build a TC. Each value set for those controls in the user interface maps to the execution of one or two scripts at build-time (preprocess script, post-process script or both).

To enable the support for build variants, set-up the preference *UML Development – Real Time C++ Transformations – Use build variants for build configuration – Build Variants* to reference a **build variants script**. This script will be interpreted by RSARTE when an interactive build is performed. It can use the BVF API to contribute a custom user interface where choices can be made for defining which variant of the application that should be built. For example:



Let's look at an example of what such a build variants script could look like in order to provide a user-interface with two controls (a checkbox and a dropdown menu) that appears when a TC is built.

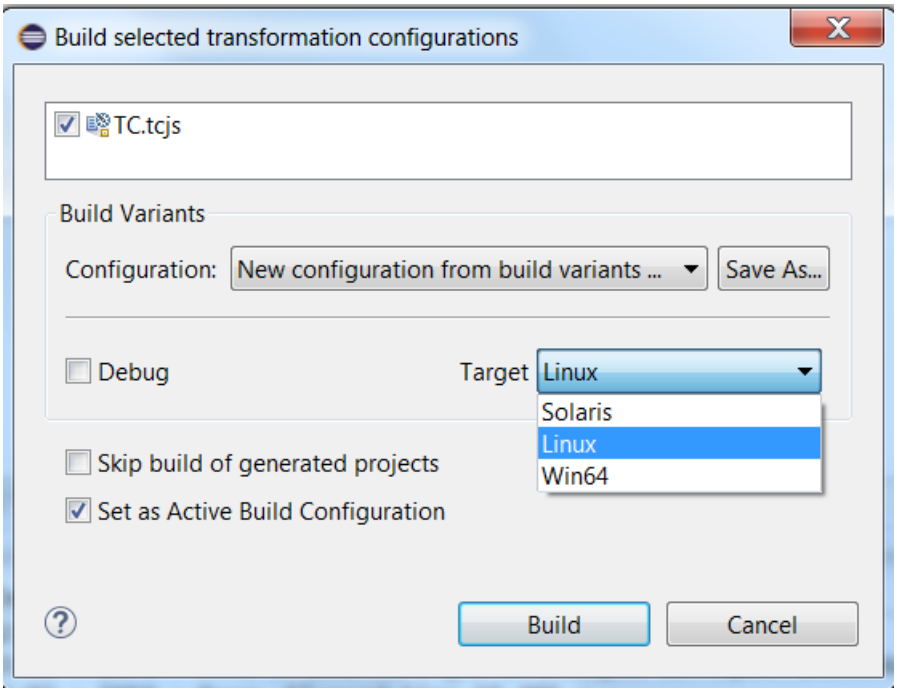
```
let debug = {
   name: 'Debug',
   script: 'debug.js',
   control : { kind: 'checkbox' },
   defaultValue : false,
   description: 'Build for debugging'
};

let target = {
   name: 'Target',
   alternatives: [
    {
      name: 'Solaris',
      script: 'Target.js',
      args: [ 'Solaris' ],
      description: 'Settings for Solaris target platform'
    },
    {
      name: 'Linux',
      script: 'Target.js',
      args: [ 'Linux' ],
      defaultValue: true,
      description: 'Settings for Linux target platform'
    },
    {
      name: 'Win64',
      script: 'Target.js',
      args: [ 'Windows' ],
      description: 'Build settings for Windows 64bit'
    }
   ]
}

function initBuildVariants(tc) {
  BVF.add(debug, target)
}
```

The function `initBuildVariants` is called when a TC is built (either from the RSARTE user interface or from the command-line). It is responsible for creating objects that represent the build variants and adding them by calling the function `add()` on the predefined BVF object. The build variant objects define two controls in the user interface. The user interface controls are defined using JavaScript objects (`debug` and `target`). Some properties of such an object define what the user interface control should look like (for example, if it should be a checkbox or a drop-down list). Other properties define what should happen during the build when that particular build variant is enabled. In the above example, the build variants script contribute one "Debug" checkbox and one "Target" drop-down menu to the user interface. Each of these controls is bound to the execution of a build variant script by means of the "script" property.

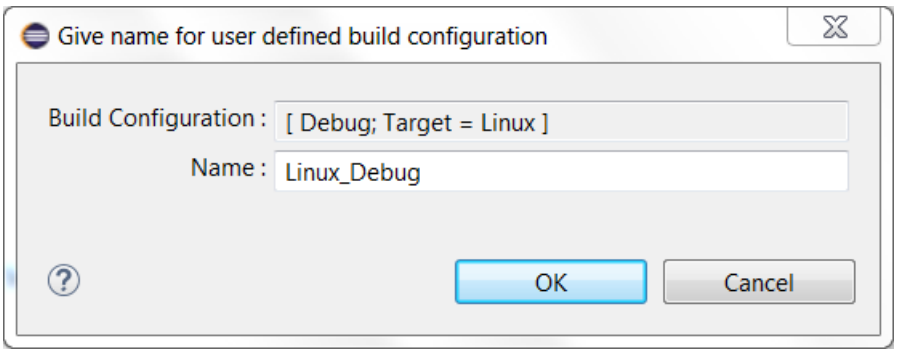
Here is what the user interface will look like when you build a TC with this build variants script enabled:



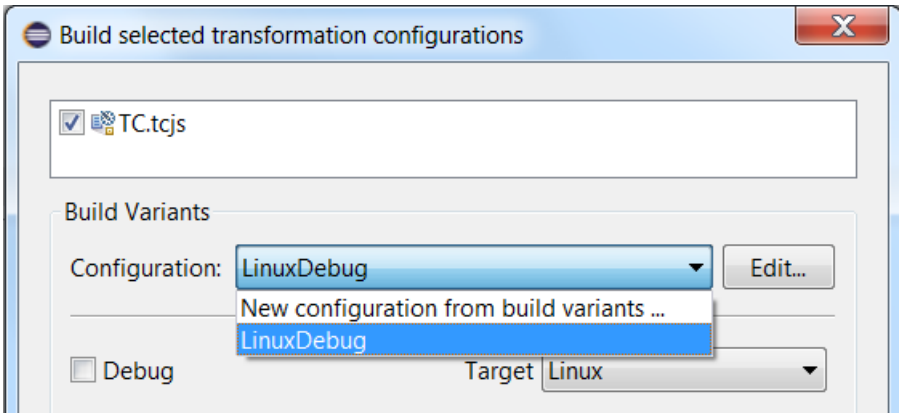
The controls after the separator in the Build Variants group are contributed by the build variants script. When pressing the Build button the TC will first be processed in the usual way. Then the scripts associated with the user interface controls will be invoked so they can modify the TC properties in order to achieve the specified build customization. If choices are made according to the picture above, the script "Target.js" will be invoked with the array ["Linux"] as argument (according to the "args" property for the Target object).

Each combination of choices made in this dialog define a set of enabled build variants. We call that set a **build configuration**. Even with only the two simple controls of the above example, we have already defined 2 * 3 = 6 possible build configurations. Each build configuration builds one specific variant of the application.

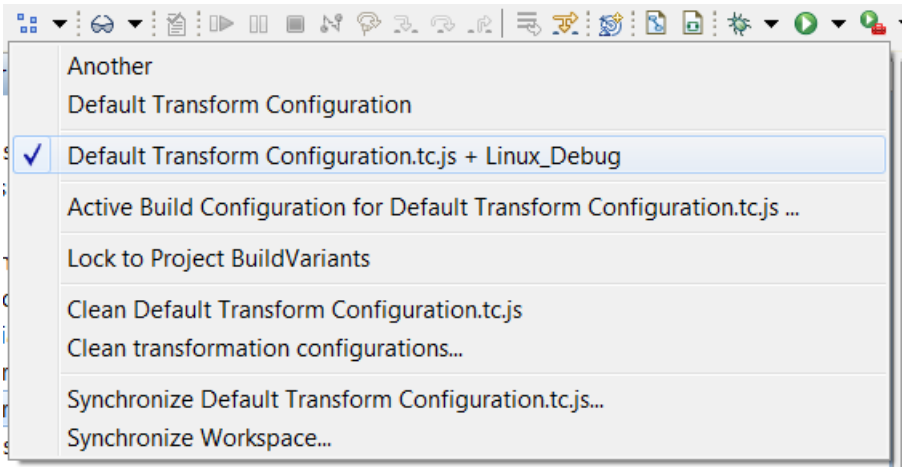
If you need to build a particular build configuration more frequently than others you can save it by pressing the "Save As" button in the dialog. Give the build configuration a descriptive name. For example:



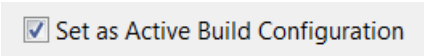
Now you can build this particular build configuration simply by choosing it from the list of build configurations in the dialog.



Named build configurations are also listed in the Build Active TC button menu so that you can build them quickly. The menu shows both which TC that will be built, and which build configuration that will be used for building it.



You can change the active build configuration either by means of the "Active Build Configuration for ..." command in this menu, or by marking the checkbox in the build dialog:



When you build a TC with an active build configuration the dialog with the build variants user interface does not pop up. This means that once you have decided which build configuration to use, you can build your TCs in the normal way without having to bother with any extra steps.

Each build configuration has a textual representation that consists of a semicolon-separated list of build variants, where each boolean build variant (checkbox in the user-interface) is identified by its name, and each enumeration build variant (drop-down menu in the user interface) is identified by its name followed by an equal sign and then followed by the name of one of its alternatives. For example: "Debug; Target=Linux". You can use this textual repres-

entation to specify which build configuration to use when running a command-line build. Use the `--buildConfig` parameter when invoking the model compiler.

Build Variant and Transformation Configuration Framework APIs

To be able to write a build variants script and the scripts it references, you need to have a basic knowledge of JavaScript. Being widely used in many applications domains it is easy to find good tutorials about JavaScript on the web. Here is one example.

In addition to the built-in JavaScript functions you will use the Build Variant Framework API for defining the available build variants, and the Transformation Configuration Framework API for working with transformation configurations inside the build variant scripts. These APIs are described in the built-in Help of RSARTE at *RSARTE Transformation Developer's Guide – API Reference - Transformation Configuration and Build Variants JavaScript API*.

External Libraries

It is common to have external C++ libraries that should be linked with the final C++ application. It may be convenient to integrate the building of such an external library into the overall build process. To do so you may use a TC which has the “Artifact type” set to “C++ External Library”. For such a TC the “Target Configuration” tab contains properties which are used so that the generated makefile can include a target for building the external library from its sources.

Generate make file for external CDT project allows the external library to be built from a CDT project. If this setting is enabled the "Build folder" property should specify a target folder for a CDT project. When the TC is built a makefile will be generated from the CDT project and the "Build command" should specify how to run make on that makefile in order to build the external library.
Note that the word "external" in the name of this property just means "external to the project that contains the TC". The CDT project is typically located in the same workspace as your model project.

Build command specifies a command that builds the external library. For example it can be an invocation of make with a makefile for the library.

Build folder specifies the folder where to run the build command.

CDT configuration name is only applicable if the setting "Generate make file for external CDT project" is enabled. In that case it can be used for specifying the name of the CDT configuration to build. If the "Build folder" already specifies a target folder for the CDT project that has the same name as the configuration (which is typical), then you don't have to specify the name of the configuration here. It will be deduced from the "Build folder" setting instead.

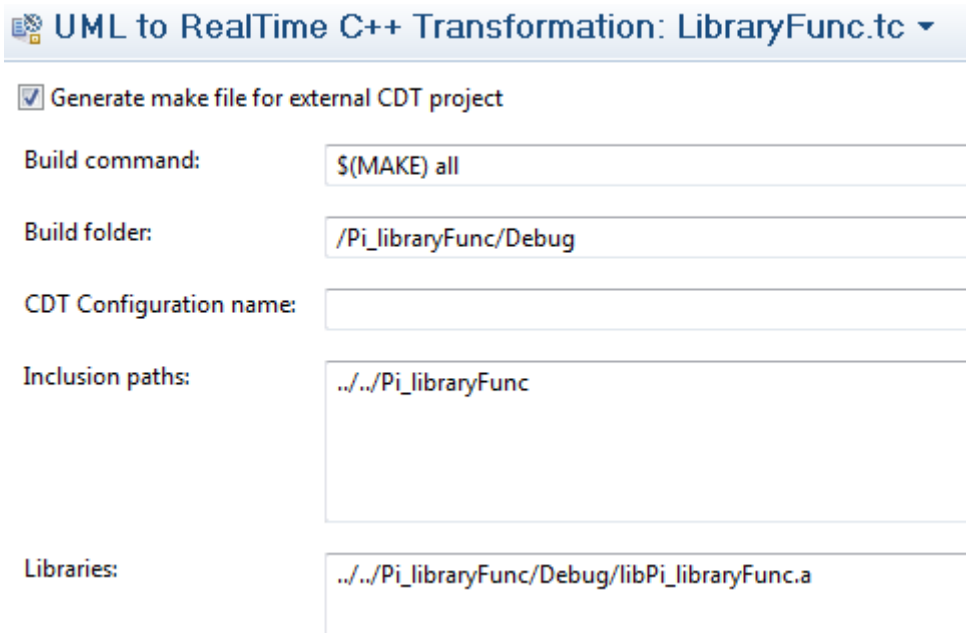
Inclusion paths is a list of inclusion paths where the header files of the external library are located. These paths will be added to the include paths in the makefile that is generated for a TC which has the external library TC as a prerequisite.

Libraries is a list of libraries that will be added to the list of libraries in the makefile that is generated for a TC which has the external library TC as a prerequisite. Typically you would here enter the libraries that are produced by running the build command.

External library TCs also have another special property **Include file name** in its "Properties" tab. It specifies an include file to be included by code that uses the external library. At most one include file can be included for the library. If you need to include more than that you have to create a wrapper include file which includes all the required files. The include file for the external library will be included by the unit header (by default called `UnitName.h`) for each TC which has the external library TC as a prerequisite TC.

Note that an external C++ library TC does not specify a target project.

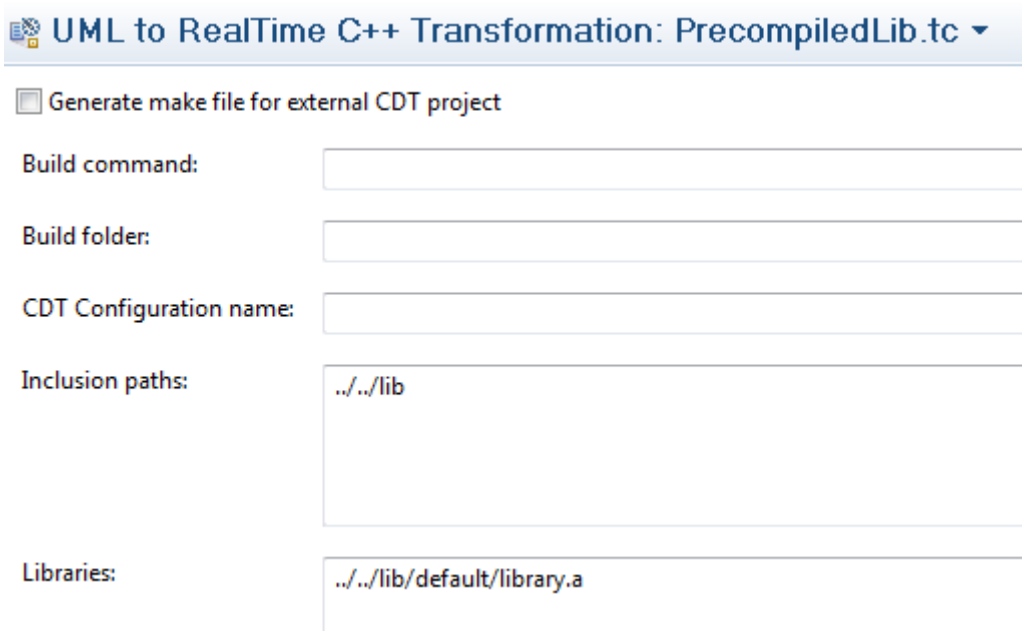
The picture below shows an example of a C++ External Library TC which is configured to build the library from a CDT project.



When this TC is built a makefile is first generated for the CDT project Pi_libraryFunc which should be available in the workspace. The CDT configuration "Debug" will be used, since that target folder is specified in the "Build folder" setting. The makefile is processed using the build command "$(MAKE) all" which will build the library "libPi_libraryFunc.a".

Precompiled Libraries

Some external libraries are rarely modified. You may want to avoid to rebuild such libraries each time you build your model, and instead link directly with a precompiled library. The precompiled library is stored in a location that is available to everyone who has to build the model. To achieve this you may leave the "Build command" and the "Build folder" settings empty in the "C++ External Library" TC. For example:



When you build an executable TC that has this PrecompiledLib.tc as a prerequisite no actions will be performed to build the precompiled library. Instead the library specified in "Libraries" will be directly used when linking the executable. This makes the build of the executable TC faster, since the library does not have to be built.

If the precompiled library is built from a UML model you may specify model elements in its "Sources" property. The code generator will automatically generate #includes for all classes mentioned there, if the setting **Generate class inclusions** is enabled. The #includes are generated in C++ implementation files that are generated from TCs that have the external library TC as a prerequisite, and that contain model elements which use the source classes.

You also need another TC (a "C++ Library" TC) which can be used to build the precompiled library, in case the model has been changed. A typical workflow is that one person changes the model for the precompiled library and then builds it using the "C++ Library" TC. Then he takes the resulting library file ("library.a" in the above example) and puts it in the location which is specified in the "Libraries" setting of the precompiled library TC. This location is typically a shared network folder, or a location in the CM system. Thereby the precompiled library becomes available for other users who need to build a model that uses it.

External Constants

By using different ”C++ External Library” TCs in different builds it is possible to build multiple variations of an application. The interface of the external library remains the same, but its implementation can vary between different builds, for example depending on target platform or build configuration (debug / release etc).

Sometimes it's not enough to only let the implementation of a library differ. The different variations of an application may also need to be built using different values for certain constants that are used in the model. For example, the multiplicity of a replicated port may be

specified by means of a constant that should have different values in different variations of the application. For this usecase external constants can be used.

An external constant is a constant that is used in the model, but its value is not defined in the model. Instead, the value is defined using a ”C++ External Library” TC (and can therefore be different for different builds of the application).

The “Target Configuration” tab contains properties that can be used for defining the values of external constants that are used in the model. The easiest way to define the values is to write them in the **Constants** field using a simple “NAME = VALUE” syntax. The VALUE can be any constant expression, and it will be evaluated by the C++ code generator when building the model. The expression may contain references to user-defined variables (see User-defined Variables), and you may use C++ style comments (// …) as necessary. For example:

```
POOL_SIZE = 100
// MYSIZE is a user-defined variable
PORT_BUF = $(MYSIZE) * 2
```

It's also possible to provide values for the external constants in a file. The file can either be a C/C++ header file (with the file extension .h) or it can be a plain text file (with any other file extension). In the latter case the text file should use the same syntax as is used in the Constants field in the TC. For a C/C++ header file regular C/C++ syntax should be used, and constants can then be defined either as macros or const definitions. For example:

```
// Constants defined in a C/C++ header file
#define POOL_SIZE 200
const int FILE_CONST = 13;
```

A file that defines external constants is referenced from the Constants field in the “C++ External Library” TC. You may either use an absolute or a workspace relative path. In the latter case the path should be preceded by the character '@'. For example:

```
// Workspace relative path
@/SomeCppProject/const.h
// Absolute path
C:\temp\constants.txt
```

If the same constant is defined multiple times, then the last definition will be used to obtain its value.

A “C++ External Library” TC can bring in values for external constants that are defined in another “C++ External Library” TC by adding that TC as a prerequisite. If necessary it can override the values for some of those external constants by providing different values for them.

Defining External Constants Programmatically

A “C++ External Library” TC has a setting **Constant provider** which can refer to a plugin in order to provide values for external constants programmatically. The plugin should have a class with a static method that will be called to obtain the values for the external constants. The method should have the following signature:

```
public static Map<String, String> void NAME(
      ITransformContext context, // current transformation context
      String transformURI,       // URI of TC
      IProgressMonitor monitor)  // current progress monitor
```

The implementation of this method should return a map which provides the names of the external constants and their corresponding values.

You should refer to this method from the “Constant provider” field using the syntax

```
<qualified plugin name>/<qualified class name>/<method name>
```

For example:

```
com.acme.extConstantPlugin/com.acme.ExtConstantClass/getExternalConstants
```

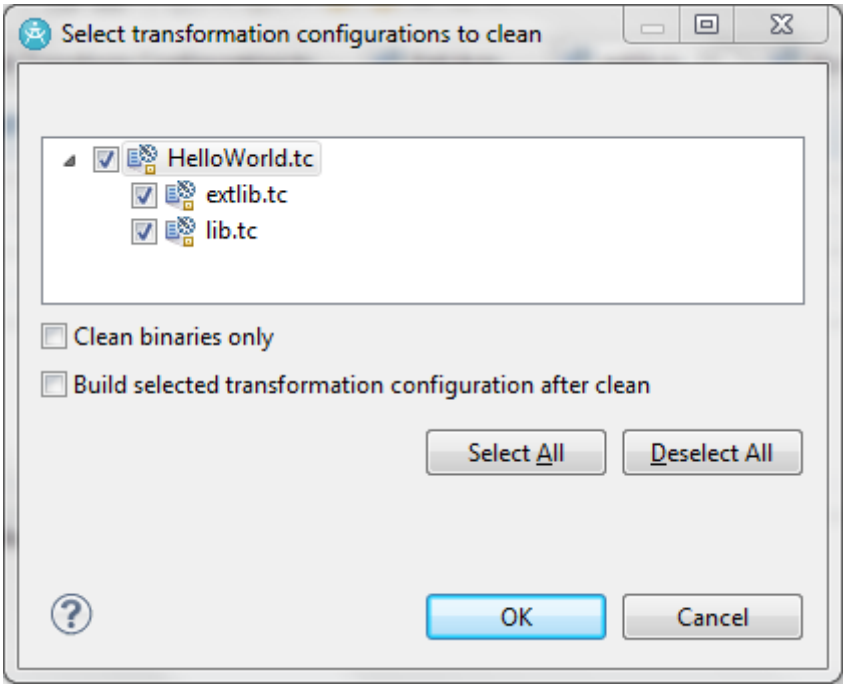
Note that constant values provided by a constant provider plugin override values provided by the Constants property.

Clean

Related to building a TC is the ability to clean it. Cleaning a TC can be done in two ways:

1. By removing the entire target project for the TC. All files that were generated when building the TC will be deleted, both source files and binaries.
2. By cleaning the target project for the TC. This removes all files (typically binaries) that were produced when building the target project. But the target project itself and the source files it contains will not be deleted.

To clean a TC invoke the ”Clean...” command in its context menu. The following dialog will appear:

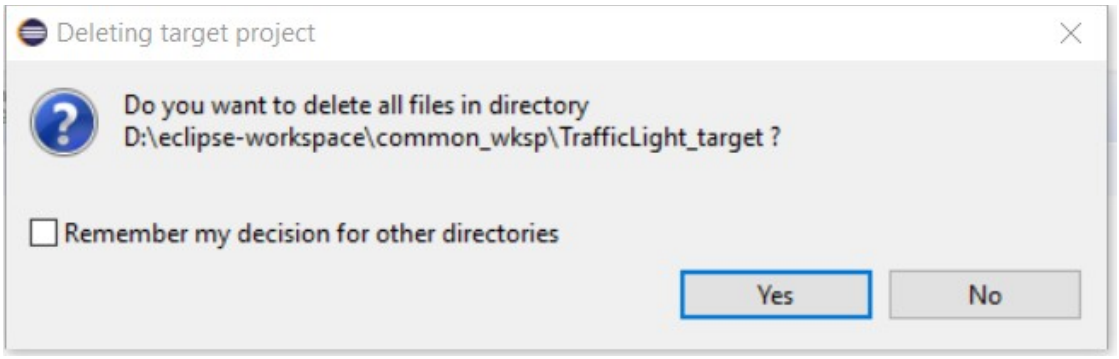


By default cleaning a TC will also clean all its prerequisite TCs, both direct and indirect prerequisites. You can unmark the checkbox for a prerequisite TC if you don't want to clean it.

When you press OK all marked TCs will be cleaned by removing their target projects. If you want to keep the target projects and generated source files and just clean the binaries, you can mark the checkbox ”Clean binaries only”. Then the TCs will be cleaned by cleaning their target projects. For a target project that has a generated makefile this means that `make clean` will be called on that makefile. Note that there is a preference *UML Development – Real Time C++ Transformations - Generate explicit clean calls for object files* which controls how the clean command is performed in the generated makefile. If you get problems with too long command lines (in case cleaning should remove a large number of object files) then set this preference to generate explicit clean calls for the object files (this is the default behavior on Windows).

Note that a TC that does not have a target project cannot be cleaned. This is the case for external C++ library TCs. If you attempt to clean such a TC nothing will happen. RSARTE does not know how to clean an external library, and you have to do this manually (for example by cleaning the CDT project that builds the external library, or to invoke `make clean` on its makefile).

If, for some reason, the target project of a TC does not exist in the workspace, but it exists on the file system, then a dialog will appear when cleaning the TC:

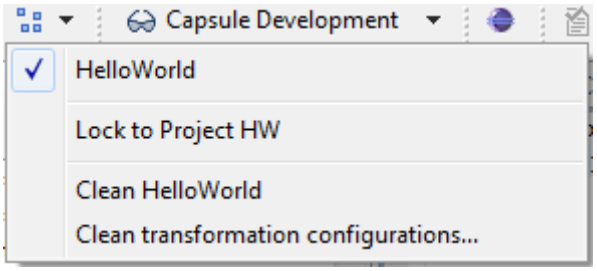


If you answer Yes, the target project folder will be deleted from the file system. If the target project contains subfolders there will popup one such dialog for each subfolder. To apply the same choice to all subfolders (either delete the subfolders or keep them) you can mark the checkbox to remember your decision before you press the Yes or No button.

In the Clean dialog there is also an option ”Build selected transformation configuration after clean”. If you select this option then the selected TC (the top-most TC shown in the dialog) will be built immediately after the cleaning is finished.

In the context menu of an RT model project there is a command ”Clean transformation configurations...” which can be used in order to clean all active TCs in that project. The same dialog as shown above will appear, and each active TC will appear as a top-level node in the hierarchies of TCs shown.

The clean command is also available as a menu choice in the ”Build Active Transformation Configuration” button menu.



From here it will clean the active TC (the one with a checkmark in front of it). If the button has been locked to a project the clean command will clean the active TC in that project. If there is no active TC set a dialog will popup to let you specify which TC to activate and clean.

As can be seen in the picture above the "Clean transformation configurations" command is also available in this menu. It is useful to invoke it from here when you want to clean TCs from many or all of the projects in the workspace. This is more convenient than to select all the projects in the Project Explorer and then invoke the command from the context menu.

As discussed in Alternative Ways to Trigger an Interactive Build Eclipse provides a common user interface for performing build of projects. The conclusion there was that this common user interface provides no benefits compared to building TCs directly using the ”Build Active Transformation Configuration” button. In the same way Eclipse provides a common user

interface for cleaning (*Project – Clean...*). If you attempt to clean an RT model project using this user interface nothing will happen.

TCs with many target projects that contain lots of source files may take a long time to clean. Performance can be improved by using these preferences in *UML Development – Real Time C++ Transformations – Clean options*:

- **Hide UI views during clean of transformation configuration**
  If this preference is enabled certain Eclipse views will be hidden during the clean operation. For example, the Project Explorer will be hidden. This improves performance by reducing the number of notifications that need to be handled when files in the target project get deleted. When the clean operation has completed, the hidden views will be automatically restored.
- **Disable indexer when number of generated projects is greater than N**
  Actions that affect a large number of generated C++ files may take a significant amount of time for the CDT indexer to handle. If you enable this preference you can specify a limit (N) for the number of generated target projects that can be cleaned with the indexer enabled. If the clean operation affects more than this number of generated target projects the CDT indexer will be disabled during the clean operation. The indexer is automatically enabled again when the clean operation has completed.