

# Building C++ Applications with DevOps Model RealTime

*Author: Mattias Mohlin  
Senior Software Architect  
IBM*

|  |           |
|--|-----------|
| <b>BUILDING C++ APPLICATIONS WITH.....</b>                     | <b>1</b>  |
| <b>DEVOPS MODEL REALTIME.....</b>                              | <b>1</b>  |
| <b>INTRODUCTION.....</b>                                       | <b>3</b>  |
| <b>TRANSFORMATION CONFIGURATIONS.....</b>                      | <b>3</b>  |
| TRANSFORMATION CONFIGURATION PROPERTIES.....                   | 5         |
| <i>Main tab.....</i>   | 5         |
| <i>References tab.....</i>                                     | 6         |
| <i>Code Generation tab.....</i>                                | 7         |
| <i>Target Configuration tab.....</i>                           | 10        |
| <i>Threads tab.....</i>  | 13        |
| <i>Code tab.....</i>   | 13        |
| DYNAMIC PROPERTIES IN TRANSFORMATION CONFIGURATIONS.....       | 15        |
| <i>Pre-defined Variables.....</i>                              | 15        |
| <i>User-defined Variables.....</i>                             | 16        |
| <i>JavaScript Expressions.....</i>                             | 17        |
| MODEL ELEMENT REFERENCES IN TRANSFORMATION CONFIGURATIONS..... | 17        |
| CREATING TRANSFORMATION CONFIGURATIONS.....                    | 18        |
| TRANSFORMATION CONFIGURATION INHERITANCE.....                  | 19        |
| PREREQUISITE TRANSFORMATION CONFIGURATIONS.....                | 21        |
| ACTIVE TRANSFORMATION CONFIGURATIONS.....                      | 21        |
| MANAGING THE SOURCES OF A TRANSFORMATION CONFIGURATION.....    | 21        |
| <i>Organize Sources.....</i>                                   | 22        |
| <i>Detect Source Dependencies Automatically.....</i>           | 23        |
| <i>Context Sensitive Library Builds.....</i>                   | 24        |
| <i>Automated Source Management and External Code.....</i>      | 25        |
| <b>BUILDING GENERATED CODE.....</b>                            | <b>26</b> |
| MAKEFILE GENERATION.....                                       | 27        |
| <b>INTERACTIVE BUILD.....</b>                                  | <b>28</b> |
| BUILD MESSAGES.....  | 29        |
| BUILDING MULTIPLE TRANSFORMATION CONFIGURATIONS.....           | 31        |

|  |           |
|--|-----------|
| ALTERNATIVE WAYS TO TRIGGER AN INTERACTIVE BUILD.....              | 31        |
| <b>BATCH BUILD.....</b>  | <b>32</b> |
| <b>PERL CONFIGURATION.....</b>                                     | <b>32</b> |
| <b>MODEL COMPILER VALIDATION RULES.....</b>                        | <b>33</b> |
| <b>BUILD VARIANTS.....</b>   | <b>33</b> |
| BUILD VARIANT AND TRANSFORMATION CONFIGURATION FRAMEWORK APIs..... | 38        |
| DEBUGGING BUILD VARIANT SCRIPTS.....                               | 38        |
| <b>EXTERNAL LIBRARIES.....</b>                                     | <b>38</b> |
| PRECOMPILED LIBRARIES.....   | 40        |
| EXTERNAL CONSTANTS.....  | 42        |
| <i>Defining External Constants Programmatically.....</i>           | <i>43</i> |
| <b>CLEAN.....</b>  | <b>43</b> |
| <b>CODE PREVIEW.....</b>   | <b>45</b> |
| USING CODE PREVIEW FOR CODE COMPARISON.....                        | 46        |
| GENERATING CODE PREVIEW FOR A TRANSFORMATION CONFIGURATION.....    | 47        |
| REMOVING CODE PREVIEW.....   | 48        |

This document describes the steps involved when building a real-time C++ application from a UML-RT model in DevOps Model RealTime.

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

The document was last updated for Model RealTime 12.0.1. All screen shots were captured on the Windows platform.

## Introduction

One of the most important capabilities of Model RealTime 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. An Eclipse 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 Model RealTime user interface, and **batch build** from command line or scripts<sup>1</sup>. 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.

The utility in Model RealTime which builds a model is called the **model compiler**. This is a stand-alone command line tool which runs as a separate application outside of the Model RealTime IDE. It can therefore be used for true batch builds that run without any dependency on the Model RealTime IDE. It can also be used for interactive builds from within the IDE. In that case Model RealTime will launch the model compiler for generating the code and a make file, and then generated code is built by make.

If your system supports parallel execution of make rules it's also possible to generate a single make file from Model RealTime 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. Also note that this feature is currently only available in batch builds.

In the following chapters we will go through different aspects that are related to building an Model RealTime C++ model with the model compiler.

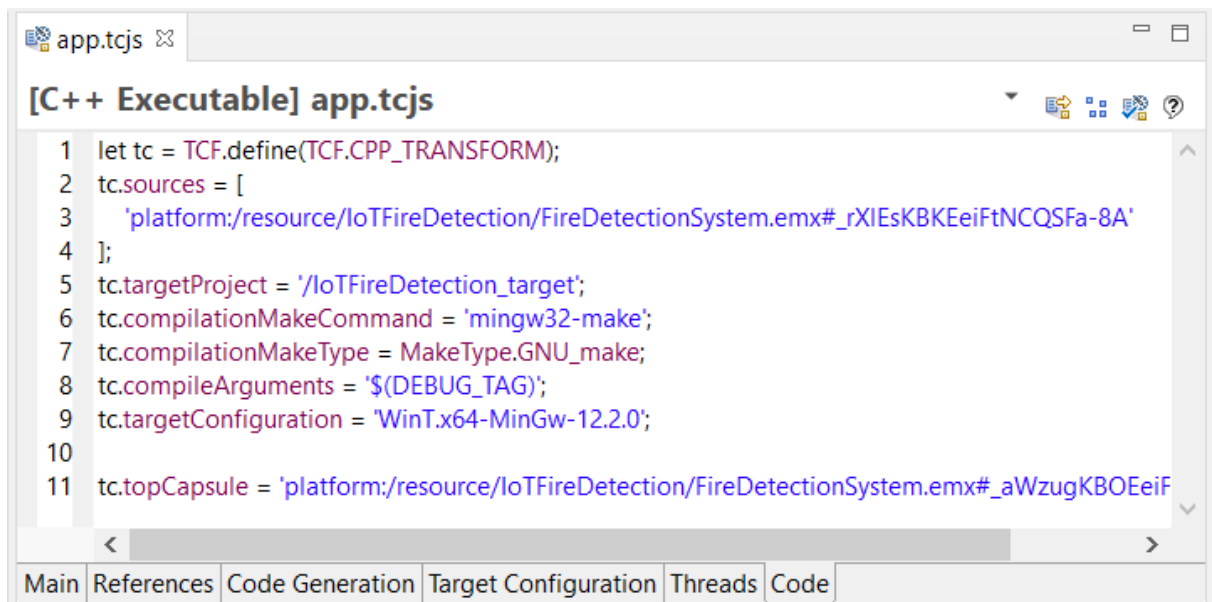
## Transformation Configurations

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.

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

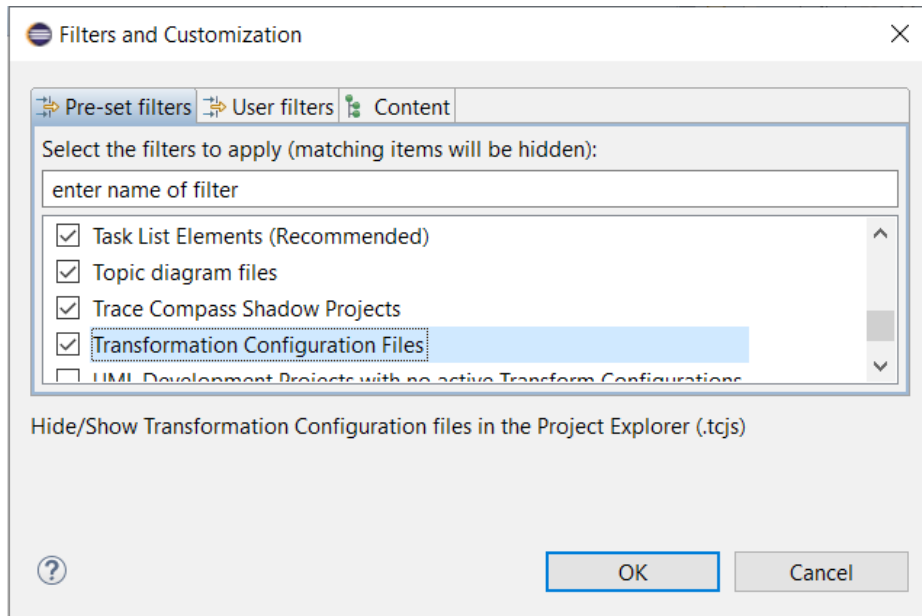
All such properties that are used in a build are stored in a **transformation configuration** (or TC for short), which needs to be specified when starting the build. A TC is a text file in JavaScript format (with the file extension .tcjs). Using JavaScript for defining build properties has many advantages. For example, it allows for dynamic properties where the value is not a static value but computed by JavaScript code. It is also a readable format that is easy to edit with any kind of text editor. There are good editors for JavaScript available for Eclipse that can be installed on top of Model RealTime. However, if you don't plan to use powerful JavaScript constructs in your TC files, you can use the TC editor that is included in Model RealTime. This is an editor that groups TC properties in logical tabs and allows you to view and edit the properties in graphical forms. However, it also provides a Code tab where you can see the JavaScript representation of all properties that have been set.



```
app.tcjs
[C++ Executable] app.tcjs
1 let tc = TCF.define(TCF.CPP_TRANSFORM);
2 tc.sources = [
3   'platform:/resource/loTFireDetection/FireDetectionSystem.emx#_rXIEsKBKEeiFtNCQSfa-8A'
4 ];
5 tc.targetProject = '/loTFireDetection_target';
6 tc.compilationMakeCommand = 'mingw32-make';
7 tc.compilationMakeType = MakeType.GNU_make;
8 tc.compileArguments = '${DEBUG_TAG}';
9 tc.targetConfiguration = 'WinT.x64-MinGw-12.2.0';
10
11 tc.topCapsule = 'platform:/resource/loTFireDetection/FireDetectionSystem.emx#_aWzugKBOEeiF'
```

Main | References | Code Generation | Target Configuration | Threads | Code

By default TC files are not shown in the Project Explorer, and instead TCs are shown under a virtual folder called "Transformation Configurations". 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":



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

## ***Transformation Configuration Properties***

If you double-click on a TC file 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 properties. In this chapter we go through the TC properties as they appear in the TC editor.

### **Main tab**

This tab contains the main properties that must be set-up for all kinds of TCs.

**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.

**Environment** is usually set to "TargetRTS". However, if none of the generated C++ files will depend on anything from the RT services library you may instead set it to "Standalone". In that case the application will not link with the TargetRTS library and hence be smaller.

**Sources** specify which elements that will be transformed to C++ when building the TC. 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 only source element. 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 and compiled, if the package contains elements that do not need to be translated 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 **Target** properties specify the name and location of an Eclipse CDT project to which generated C++ files will be written. This project can either be created and updated when the TC is built, or you may use an existing project which you maintain manually. In the latter case the properties on the [“Target Configuration” tab](#) are 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. Use then the **Workspace output path** property to specify the location of the CDT project and all generated code. You may use an absolute path for this property, but more commonly a relative path which then will be resolved against the **Location** property (which by default is the location of the workspace). Note that the name of the CDT project (either existing or generated) will be determined from the last segment of the output path. Read more about the "Automatically create and update target project" property in [Building Generated Code](#).

You can document a TC by writing a comment in the **Documentation** field. The comment has no impact on building the TC.

## References tab

A TC may reference other TCs. Use the buttons in this tab to add or remove TC references. You can navigate to referenced TCs by double-clicking on them.

**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 [Transformation Configuration Inheritance](#) for more information about transformation configuration inheritance.

**Prerequisite transformation configurations** lists other TCs which are "prerequisites" of this TC. When a TC is built, all its prerequisite TCs will first be built. This can for example be useful when building an executable TC that links with libraries built by other TCs (added as prerequisites). See [Prerequisite transformation configurations](#) for more information.

The relationships to other TCs that you create using the References tab are important in order to understand what happens when the TC is built. If there are many TCs and relationships it can be useful to visualize this graphically. You can do this by means of the Project Explorer context menu command **Visualize - Explore in Graphs**. Read more about this feature in the built-in help topic *Model RealTime User's Guide – Articles – Editing – Diagrams – Visualizing TC relationships graphically*.

## Code Generation tab

This tab contains all properties that control how to translate the source elements of the TC to C++ code.

**Top capsule** is a property that is only available for an executable TC. 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.

**Compile data classes individually** controls how data classes (i.e. passive classes) are compiled. By default this property is set, 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.

**Generate code qualifiers** 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#_xj1dcPkAEeGEhK1G362qaA
log.log("Hello World from C++ Capsule");
context()->abort();
//}}}}USR
```

**Generate fully qualified state names** can be useful to set if you have hierarchical state machines where the same state name is used more than once. The `rtg_state_names` array in generated code will then contain fully qualified state names which can make debugging and trouble-shooting easier.

**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 generated `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)
```

The property **Context sensitive library build** 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.

**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:

Default arguments:

**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.

**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.

**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.

**Capsule factory** can be used for specifying a global capsule factory which will be used for creating and destroying capsule instances in case no more specific capsule factory is specified on a certain capsule part. You should specify a C++ expression here that evaluates to an `RTActorInterface*`. To learn more about capsule factories, see the article "Custom capsule constructors" in the Model RealTime documentation. You can find this document in the built-in Help under *Model RealTime User's Guide – Articles – Modeling realtime applications*.

**C++ code standard** specifies the C++ code standard which the generated code will comply with. By default the code standard is specified as a workspace preference (*RealTime Development – Build/Transformations – C++ – C++ code standard*) and you only need to set this TC property if you want to override the code standard for a particular build.

**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:

Copyright text:

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

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

## Target Configuration tab

This tab contains properties that control how to generate the makefile to be used for building generated C++ code. Note that if the “Target” properties on the [“Main” 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. Also, if the “Artifact type” property on the [“Main” 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.

**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 location in the Model RealTime installation where the C++ implementation of the RT services library resides. If you have your own version of the RT services library, enter the path to it here.

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 Model RealTime instead, which is improved in many ways, you should therefore update this property 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.

**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)

If this property is unset (or set to Default) 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. Or use `$(DEBUG_TAG)` as a compile argument which will be expanded to the correct debug flag depending on compiler used.

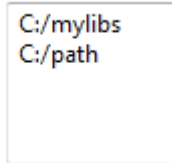
**Compile command** is by default set to \$(CC). This variable expands to the compiler to use, which follows from the property "TargetRTS configuration". If you want to use a different compiler than the one that is by default used for the selected TargetRTS configuration, then you can change this property.

**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:

Inclusion paths:



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:

```
O_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 arguments** specifies additional arguments for the linker.

**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 (*RealTime Development – Build/Transformations – C++ – 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.

**Top make command** and **Top make arguments** 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 workspace preference *RealTime Development – Build/Transformations – Type of Generated Make Files* 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.

**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 <Model RealTime INSTALLATION>/rsa\_rt/C++/TargetRTS/libset/default.mk.

**User libraries** allows you to specify custom libraries to pass to the linker. Type each user library file 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 contains properties that control which threads to use in the generated application, and how to map 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:

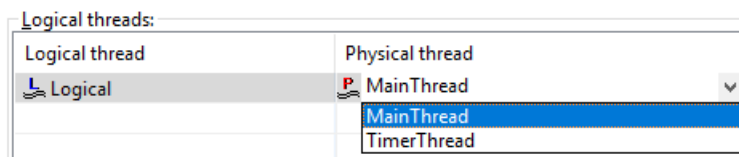
- **MainThread**  
This is the thread which by default runs all capsule instances in the model.
- **TimerThread**  
This thread is used by the Timing service of the RT services library to implement timers. It is always present even if you don't use timers in the model.

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.
- **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.
- **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.
- **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:

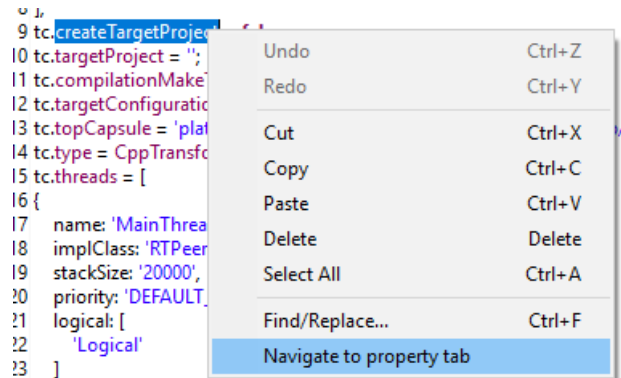


## Code tab

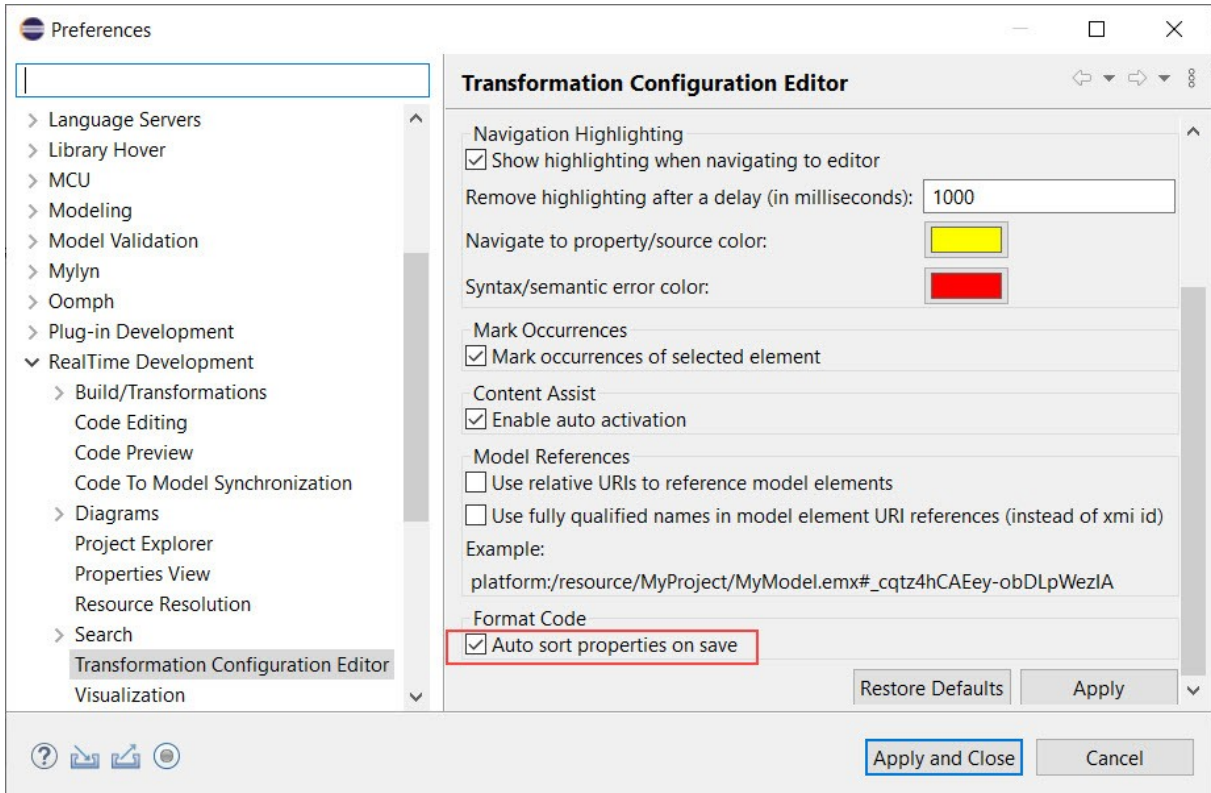
As already mentioned, this tab shows all properties that have been set in the TC. Each property with a value different from its default value is present. The properties are shown in JavaScript syntax and you can edit them as needed. Note, however, that the Code tab does not provide a full-fledged JavaScript editor. If you plan for extensive textual editing of TCs, you

may prefer to install a dedicated JavaScript editor into Model RealTime, or edit them with an external JavaScript editor.

You can navigate from properties shown in the Code tab to the corresponding properties shown in the other tabs. Right-click on a property in the text and perform the command "Navigate to property tab" shown in the context menu.



The Code tab includes a useful feature that automatically sorts properties upon saving a TC file. For this, the "Auto sort properties on save" setting in the workspace preferences at *RealTime Development - Transformation Configuration Editor* is always enabled by default. This guarantees that properties are sorted automatically when saving TC files in your UML application. Keep in mind that when this feature is active, comments at the top of a property will also be moved along with the property to its new position during sorting.



This functionality offers several benefits:

- Readability: Easier to scan and understand, especially in larger files.
- Consistency: Promotes uniform style across your project's TC files.
- Reduced merge conflicts: Lowers the risk of merge issues when multiple people edit the TC file.

## **Dynamic Properties in Transformation Configurations**

TC properties need not have static values only. Since they are defined using JavaScript you can use any JavaScript expression to define the value of a property. But even without using JavaScript, properties can become dynamic by referencing variables.

Another way to make a TC dynamic, is to let a script change it on the fly when it is built. Read more about this in [Build Variants](#).

## **Pre-defined Variables**

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 `<Model RealTime 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:

|                                |   |
|--------------------------------|---|
| <code>\$(TOP_CAPSULE)</code>   | 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.  |
| <code>\$(TCONFIG_NAME)</code>  | 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, for example in the "Workspace output path" property.  |
| <code>\$(CAPSULE_CLASS)</code> | This variable expands to the name of a class generated from a capsule, and can be used in the "Capsule factory" property. For example, you can use it for passing the capsule class as a template parameter to the specified capsule factory, so that the capsule factory can create an instance of the capsule class using the new operator. |
| <code>\$(workspace_loc)</code> | This is a standard Eclipse variable which expands to the location of the Eclipse workspace. It can be used in the "Workspace output path" property. It is also available as a make file variable and can therefore be used in all properties which appear in the generated make file (e.g. "Inclusion paths").                                |

## User-defined Variables

It is possible to define your own variables and use them within a TC. This can be useful in order to create more generic TCs 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. Path variables defined in the workspace preferences at *General – Workspace - Linked Resources*
2. String substitution variables defined in the workspace preferences *Run/Debug – String Substitution*
3. Environment variables defined in the system

Most variables do not need to be resolved until the TC is built by the model compiler. If a referenced variable is not defined in any of the three contexts listed above, the model compiler will print a warning. The build will still proceed, but may fail later since the variable could not be resolved. It is therefore strongly recommended to pay attention and fix such warnings. Here is an example of what the warning will look like:

```
WARNING : Cannot resolve variable '$(TARGET_LOC)' in 'Location' property:'$(TARGET_LOC)'
```

Note that path variables always must specify an absolute path, and can therefore only be used in TC properties that specify paths. Here is the list of TC properties where user-defined variables can be used:

- The "Location" property in the [Main 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 (in this case either as a string substitution variable or environment variable). If the location instead would be specified as "\$(TARGET\_LOC)" then a path variable could be used instead, since the variable then can be resolved with an absolute path.
- The "Target services library" property in the [Target Configuration tab](#). Note that this variable is resolved already by the TC editor in order to populate the list of TargetRTS configurations.
- The "Build folder" property in the [Target Configuration tab](#) for a "C++ External Library" TC.
- The "Build command" property in the [Target Configuration tab](#) for a "C++ External Library" TC.
- The "Clean command" property in the [Target Configuration tab](#) for a "C++ External Library" TC.
- The "Constants" property in the [Target Configuration tab](#) for a "C++ External Library" TC. Here path variables are not applicable since the variables specify the values of constants to be used during the build. See [External Constants](#) for more information.

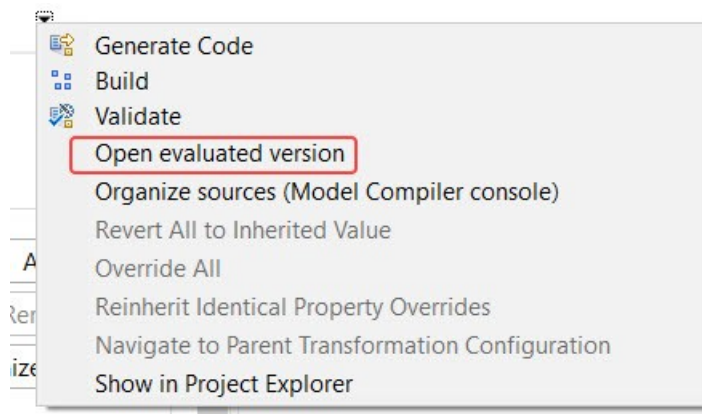


## JavaScript Expressions

The most dynamic way to specify the value of a TC property is to use a JavaScript expression. Such expressions can reference variables defined in JavaScript, contain calls to JavaScript functions and in general use all other features of the JavaScript language. JavaScript expressions are evaluated using Eclipse Nashorn. This in turn means that they can contain calls to Java APIs. For example, here is an example of how to define a TC property by dynamically reading the value of an environment variable using a Java API:

```
tc.inclusionPaths = [  
    java.lang.System.getenv("INCLUDES");  
];
```

To make it easier to work with dynamic TCs, the TC editor provides a feature for evaluating all properties of a TC and view them in a read-only TC editor. The command to use is called “Open evaluated version” and is available in the TC editor toolbar menu.



## Model Element References in Transformation Configurations

Some TC properties refer to elements in the model, for example "Sources" and "Top capsule". By default such a reference consists of a URI that identifies the model file where the element is located (relative to the Eclipse workspace folder), and the unique ID of the element. Here is an example:

```
tc.sources = [  
    'platform:/resource/xxx/CPModel.emx#_KZK5YBuoEeyDNsbmUkqd7g'  
];
```

You can navigate to the referenced element in the Project Explorer by Ctrl-clicking on the URI (which then becomes a hyperlink). You can also hover the mouse over the URI to get a popup showing the element name.

There are two workspace preferences (in *RealTime Development - Transformation Configuration Editor - Model References*) you can set to change the format of these URIs:

- **Use relative URIs to reference model elements**  
If set, the file part of the URI will be relative to the TC file rather than the workspace folder.

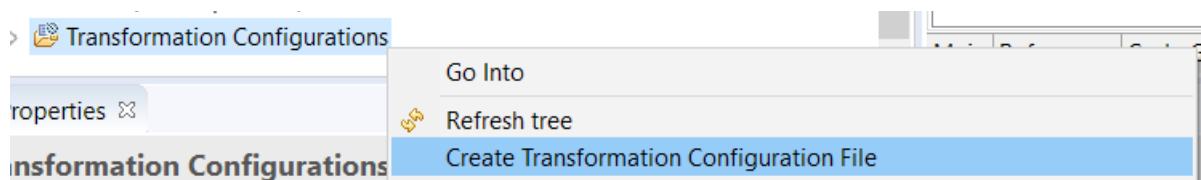
- **Use fully qualified names in model element URI references (instead of xmi id)**  
If set, the element's fully qualified name will be used instead of its unique ID.

Both these preferences make model element URI references more portable so you can copy/paste a TC file (or a project containing a TC file) without having to manually update all model element references in the copied TC. However, certain refactorings (like moving a referenced element) may then require you to update the URIs. Note that the preferences only affect what URI format to use for newly created model element references, and will not change existing references in the TC.

The navigation and mouse hover features mentioned above works regardless of how these preferences are set (but the need for these features may be smaller since with qualified names it's easier to conclude directly from the URI what element it references).

## Creating Transformation Configurations

When you create a new model project using a template in the "UML Capsule Development" category, you will get a default transformation configuration to use for building the model. 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:



You can also do it by means of the **File – New – Other – Transformation Configuration** command, or by copy/paste of an existing TC in the Project Explorer.

Before you can build a new TC you need to set-up a few properties. At a minimum you need to specify where generated code should be placed (the “Target” properties) and which elements to translate to C++ (the “Sources” property). Depending on the type of TC (“Artifact type” property) you may also need to set-up some other properties. For example, for an executable TC you must specify the “Top capsule” property (otherwise Model RealTime doesn't know which capsule to incarnate at application start-up).

You can check so that the TC is well-formed by validating it.



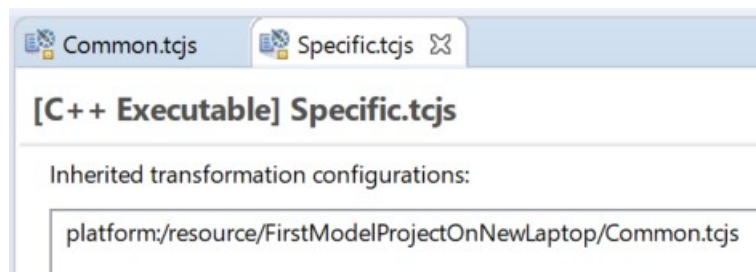
If no errors are reported, you can then generate code for the TC (that is, just generate the C++ code, not build it):



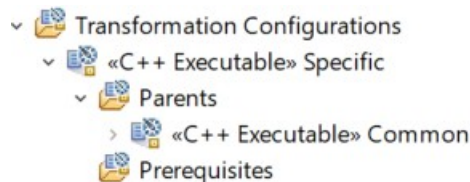
## Transformation Configuration Inheritance

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 a base TC which the other TCs inherit from. You would not build that base 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 a TC “Specific” that inherits from a base TC called “Common”:



The inheritance can also be seen in the Project Explorer (if you have set the preference *RealTime Development – Project Explorer – Show inherited transformation configurations*):



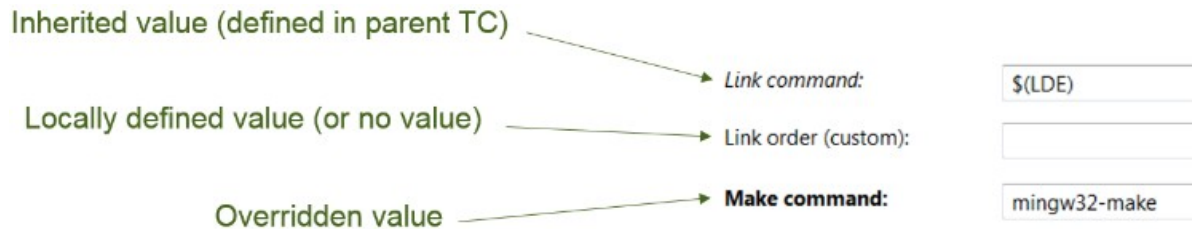
Multiple inheritance of TCs is supported, i.e. a TC can have more than one parent TC.

When you change a property of a TC that inherits from another TC, the property will be marked in boldface. This shows that the property has been overridden in the inheriting TC. For example:

Top capsule: <None>  
 **Compile data classes individually**

Note that the property may have a different or the same value in the inheriting TC as in the inherited TC. Boldface just shows that there is a value defined for the property in the inheriting 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 TC editor which help you work with inherited properties. Commands that apply for a single property are located in the context menu of that property (right-click on the property label), while commands that apply for all properties in the TC are located in the toolbar menu in the TC editor.

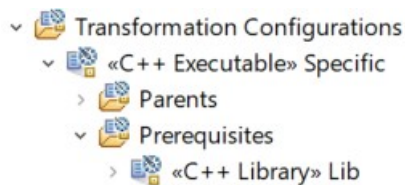
- **Revert to Saved**  
Restore the property to what is stored in the TC file. Useful in case you made some changes (not yet saved) that you want to undo.
- **Delete Value**  
Remove the overridden value for the property, so that the value stored in the inherited TC is shown instead. After this the property will appear in italics instead of boldface to show that it no longer is overridden.
- **Revert All to Inherited Value**  
Reverts (i.e. deletes) overridden values for all properties in the TC.
- **Override**  
Overrides a property by storing the inherited value in the inheriting TC file. After this the property will appear in boldface to indicate that it now is overridden.
- **Override All**  
Overrides all properties in the TC.
- **Reinherit Identical Property Overrides**  
Deletes the values for all properties that are overridden but have the same value as in the inherited 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 Source Code**  
Shows where the property is assigned its value in the Code tab.
- **Navigate to Parent Transformation Configuration**  
Similar to "Navigate to Inherited Value" 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 also 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.

## Prerequisite Transformation Configurations

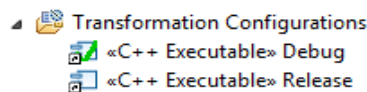
If a TC A has a TC B as its prerequisite it means that B has to be built first, before A can be built. You can use prerequisite relationships when you need to enforce a particular order in which different TCs are built. The typical example is to build an executable that links with a C++ library. The library must be built before the executable can be built, so we would use a prerequisite relationship from the executable TC to the library TC.

Prerequisite TCs are by default shown as subnodes in the Project Explorer, and you can navigate to them by double-clicking. 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 the preference *RealTime Development – Project Explorer – Show inherited transformation configurations* has been set, prerequisite TCs are shown under a "Prerequisites" node to distinguish them from inherited (i.e. parent) TCs.:



## Active Transformation Configurations

You can set a TC in your model project as active to tell Model RealTime that you want to build this particular TC when building the project. To set a TC as active you can right-click 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 in the Project Explorer. For example:



## 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.

If all elements contained in a package should be part of the executable or library, you can specify that package as the only source element of a TC. This is convenient since you then don't have to update the Sources list each time new elements are added to the package. However, there is of course also the risk that you, at some point, add some model elements to the package that only are needed at model-level, and not in the C++ application. And this results in a longer than necessary build time, and possibly also a binary that is bigger than necessary.

Model RealTime has three features that can help you optimize the build to avoid building unnecessary source elements: [Organize Sources](#), [Detect Source Dependencies Automatically](#) and [Context Sensitive Library Builds](#). All these features work by analyzing references in the model to find out what elements that need to be included in the build.

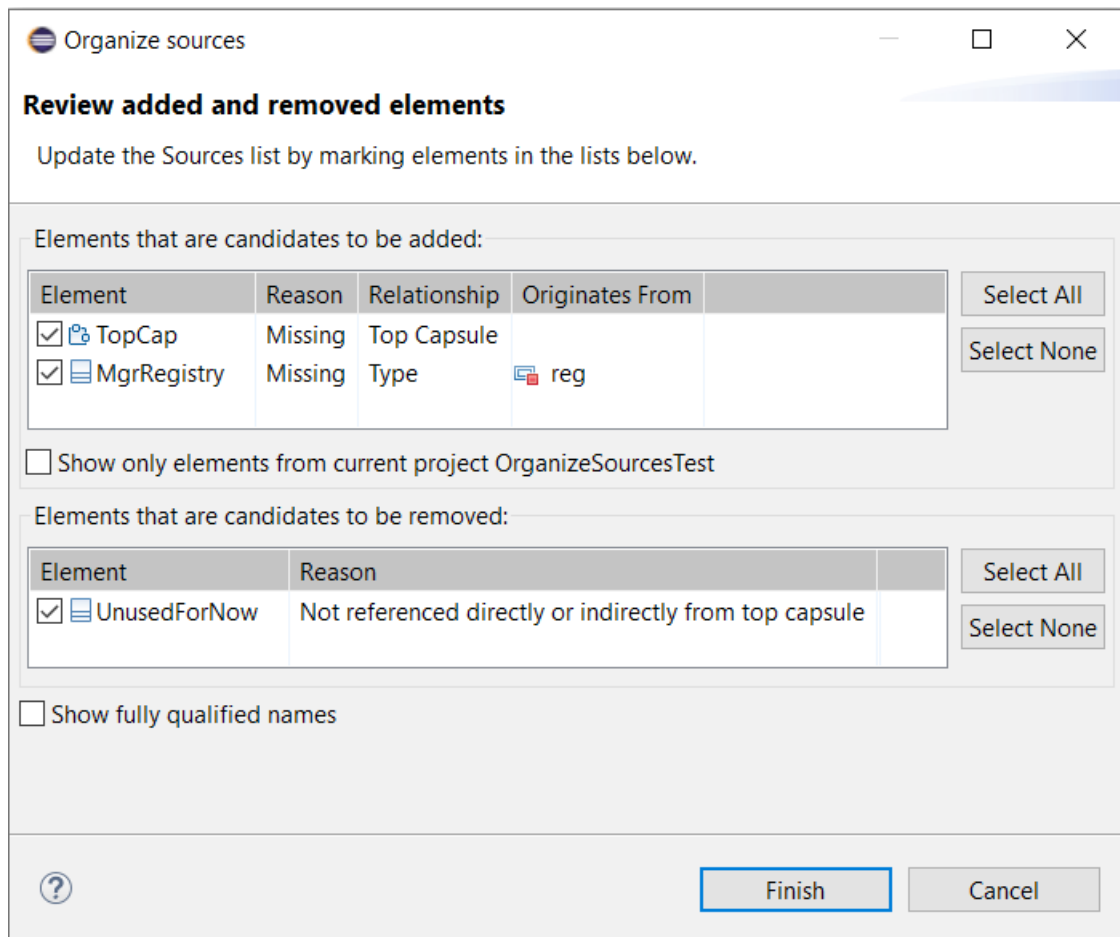
## Organize Sources

The TC editor provides a dialog that can assist you to set-up the Sources list in the optimal way. Open this dialog by pressing the "Organize Sources" button:



Before you can press this button you need to have specified the top capsule (for an executable TC) or at least one source element (for a library TC). The Organize Sources feature will use these elements as the starting point when analyzing references in the model to find out what elements that need to be built.

Here is an example of what the Organize Sources dialog may look like:



The upper table lists elements which you are suggested to add as source elements. For each element you can see the reference that is the reason for why that element needs to be a source element of the TC. If all elements in a package are needed source elements, then the package itself will be suggested to be added instead (to keep the Sources list short).

The lower table lists elements which are currently listed as source elements, but don't need to be so, because they are not referenced by other source elements or the top capsule. Here you may also find a package in case not all of its contents need to be transformed by the TC. Then the needed elements of the package are candidates to be added, and the package itself can be removed from the Sources list.

Note that Organize Sources only provides suggestions for how to update the Sources list to make it optimal. Use the checkboxes of each element to decide if you want to follow that suggestion or not. Then press the Finish button to update the Sources list.

It's important to be aware that all suggestions made by Organize Sources are made by analyzing references in the model. References in C++ files will not be included in the analysis. Also, for a library it is possible (and even common) that you want to include more elements than what is used by the library itself. Hence, it's important to carefully review all suggestions made by Organize Sources before updating the Sources list accordingly.

Sometimes an element that is referenced should still not be a source element of the TC because it's already the source element of a prerequisite TC in another project. To avoid building such an element twice you can use the checkbox "Show only elements from current project". Then referenced elements located in other projects will be filtered from the table, so you can avoid to add them. It can also help to use the checkbox "Show fully qualified names" to see where the elements are located.

If you use Organize Sources as your way of keeping the Sources element optimal, remember to invoke it at regular intervals. As you change your model, the Sources list may need to be updated again to remain optimal.

## **Detect Source Dependencies Automatically**

If the preference *RealTime Development – Build/Transformations – C++ – Detect source dependencies automatically* is set, Model RealTime will automatically detect if there are any elements that need to be included in the build, but that are not listed as source elements neither in the built TC nor in its prerequisites. When an executable TC is built, this analysis starts from the specified top capsule and any elements present in the Sources list. All elements that are referenced by these elements, directly or indirectly, will be automatically included in the build. This means that you can leave the Sources list empty for an executable TC and let Model RealTime automatically compute the source elements that need to be built.

When you build a library TC, there is no top capsule, and in that case the Sources list must contain at least one element which the reference analysis can start from.

Note that an automatically added source element will be built by the same TC that builds the element that references it. If the built TC and its prerequisites specify different build settings,

this can lead to that automatically included source elements get built with incorrect build settings. Therefore, the model compiler will print a warning in this case:

```
16:04:33 : WARNING : Found elements which are not included as source into any TC,
some of them could have been added into wrong context
```

You can ignore this warning if you know that it doesn't matter into which TC the missing elements are included.

It's important that all users who build a model agree on if the “Detect source dependencies automatically” preference should be set or not. 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, it's solely the references from elements in the built TC that determines which elements that need to be built. The source elements of prerequisite TCs will be included in the build, but their references will not be analyzed. 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 *RealTime Development – Build/Transformations – C++ – Report details about automatically added source elements*. Then messages will be printed about which elements that are considered necessary to transform, and why.

One way to use the “Detect source dependencies automatically” feature is to have it enabled, but anyway update the TC Sources list with the elements that are reported as missing. The model compiler prints a summary message of all elements it found to be referenced, but that are not listed as source elements in the TC or its prerequisites. This message is on a format that makes it possible to directly copy and paste it into the Code tab in the TC editor to update the Sources list conveniently. There are two benefits with taking the extra time to update the Sources list:

- You can decide to which TC the missing source elements should belong. They can either be added to the built TC or to one of its prerequisite TCs. As mentioned above this choice is important if the TCs use different build settings.
- It can be easier to understand what parts of the model a TC builds when it explicitly lists the source elements.

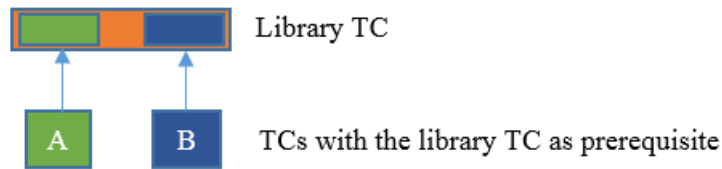
Here is an example of what this message can look like:

```
07:54:38 : INFO : Adding missing sources on-the-fly to "MyProj/HelloWorld.tcjs" :
'platform:/resource/MyProj/HelloWorld.emx#_s1xKwPoYEemJs6K8xfurHQ' /*
HelloWorld::HelloWorld */ ,
'platform:/resource/MyProj/HelloWorld.emx#_utwnIPoYEemJs6K8xfurHQ' /*
HelloWorld::MyClass */
```

## 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, Model RealTime supports a feature called “Context sensitive library builds”. It is controlled by a preference *RealTime Development – Build/Transformations – C++ – Context sensitive library builds*. 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.

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.

The "Context sensitive library build" feature can be combined with "Detect source dependencies automatically". In that case source elements of prerequisite TCs will only be included in the build of the library if they are referenced by an element in the built TC, and missing source elements will be automatically added.

## Automated Source Management and External Code

The strategies for automating the management of TC source elements described above all relies 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:

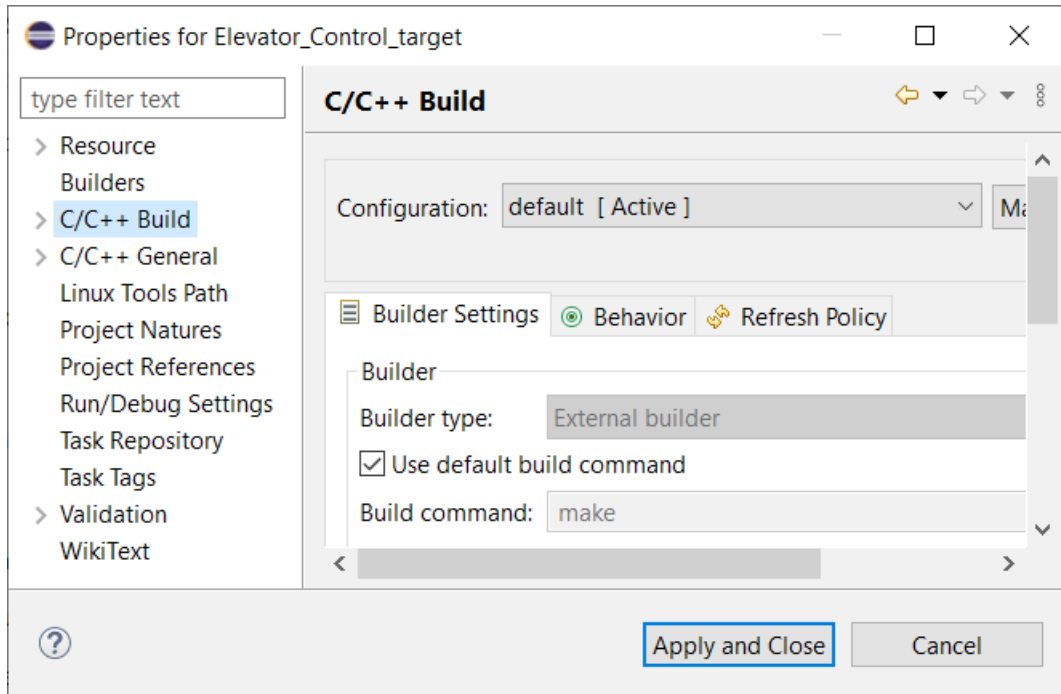
1. Create a special class in the model that contains the elements that are referenced from external C++ code. Open the Properties view and go to the "C++ General" section. Set "Generate files" to Header so that only a header file gets generated for this class.

**Generate File:**                       Implementation    Header

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 model compiler 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. This feature is, however, currently only supported for batch builds.

## **Makefile Generation**

There are two kinds of makefiles that may be generated by the model compiler (controlled by the preference *RealTime Development – Build/Transformations – Type of Generated Make Files*):

- **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.

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> <make-arguments> -f batch.mk all
```

where `<make-command>` and `<make-arguments>` are 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`.

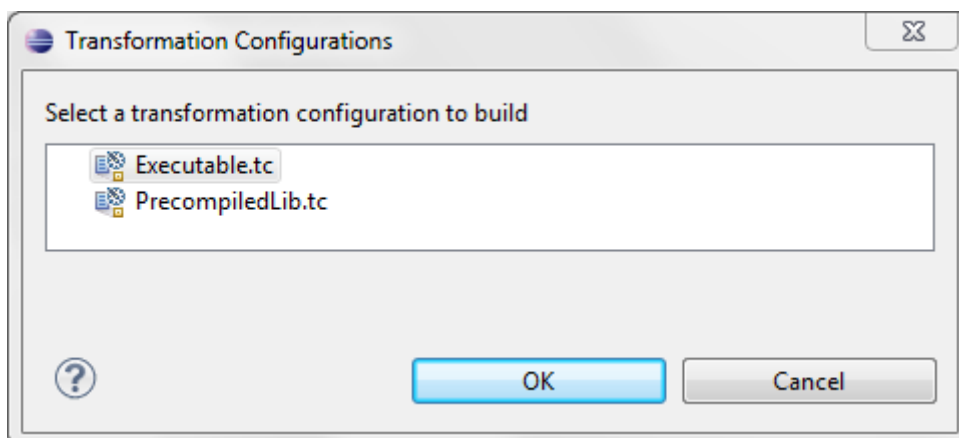
## Interactive Build

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



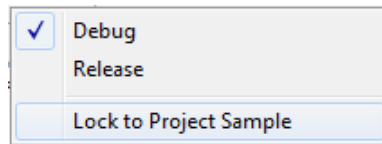
This button is always enabled and builds the active TC in the selected project. If a project is not directly selected in the Project Explorer, Model RealTime uses the project to which the selected element belongs. For example, if you select a capsule the Build Active TC button will build the active TC in the project that contains the capsule.

If the project contains 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 the project that contains that TC. 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 code generation phase.

## **Build Messages**

As soon as the build of the TC has been started, Model RealTime 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 may include:

- 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) – see [Model Compiler Validation Rules](#).
- Compiler messages
- Linker messages

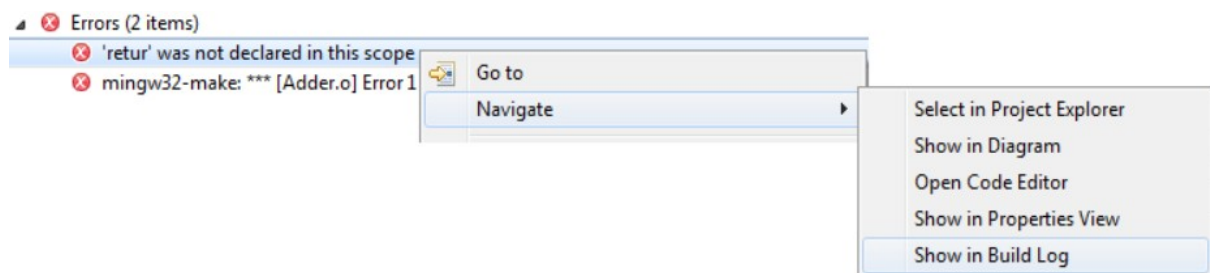
You can double-click on messages in the UML Development console in order to navigate to the location of the message. This could either be a TC, an element in the model, or it could be somewhere in the generated C++ code.

By default, compiler and linker messages (and all other messages produced by make) are printed directly to the UML Development console. If you prefer you can set the preference *RealTime Development – Build/Transformations – Show make log in UML Development Console*. Then such messages will instead be printed to a build log and a link to this 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
```

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.



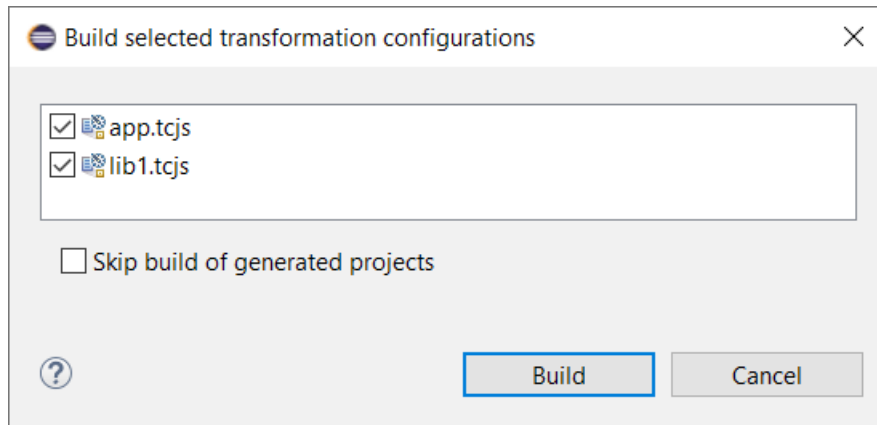
You can manually remove problems from the Problems view by deleting them. For example, you may want to delete problems that you have looked at or fixed. When you start a new build you may want to automatically delete all or some of the problems that already exist in the Problems view so that it becomes 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 *RealTime Development – Build/Transformations - 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 reappear as a result of the new build.
- **Clean problem markers for all built projects 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):



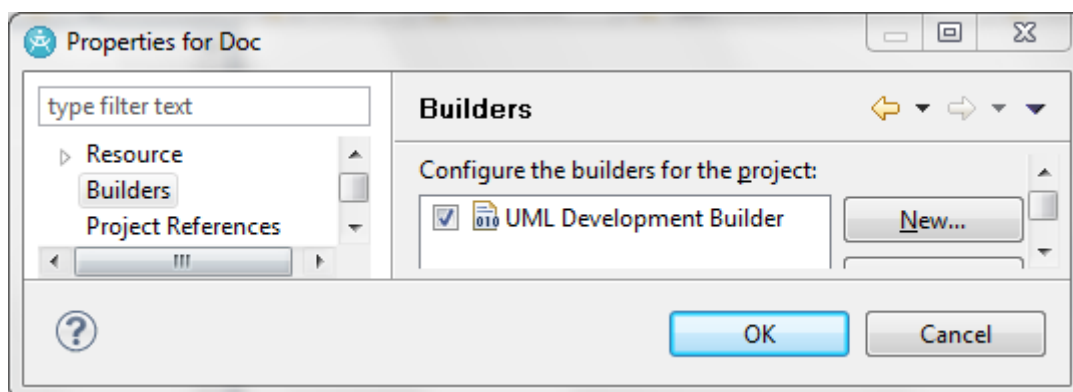
When you press Build all the marked TCs will be built one by one.

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".

## 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 Model RealTime 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 Model RealTime 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 Model RealTime C++ models (it is turned off by default in Model RealTime).

## Batch Build

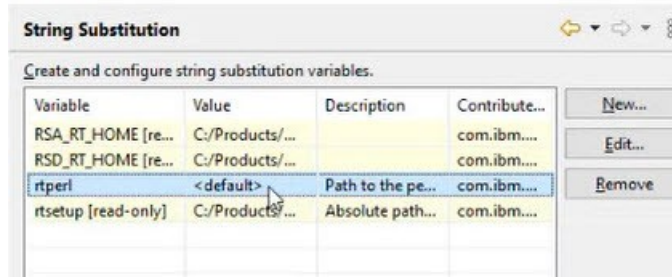
A batch build (sometimes also called headless build, or non-GUI build) can be performed by means of the model compiler. The model compiler is a stand-alone command line tool which does not have any dependency to Eclipse or something that requires a user interface. Therefore it is possible to run the model compiler 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 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 at *Model RealTime User's Guide – Articles – Building – Model Compiler*.

## Perl Configuration

Model RealTime relies on Perl for some tasks within the build process. The Model RealTime installation contains a version of Perl called `rtperl` which is used by default on Windows and Linux. It is located within the plugin `com.ibm.xtools.umldt.rt.core.tools`. If you want or need to use a different version of Perl, open *Preferences - Run/Debug - String Substitution* and edit the variable called "rtperl".





Enter the full path to your Perl executable in the "Value" field (or just "perl" if you have it accessible in your path).

**Note:** In older versions of Model RealTime the "rtperl" variable was set to the full path of the rtp Perl executable from the installation. This means that if you open a workspace created in such an old version, you should remove that path, especially if the old installation location was deleted so that path no longer is valid.

## Model Compiler Validation Rules

While transforming a model to C++, the model compiler checks the model against validation rules. To learn about the model compiler validation rules in detail, see *User's Guide – Articles – Building – [Model Compiler Validation Rules](#)* in the built-in Help.

## 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. Few examples are given below:

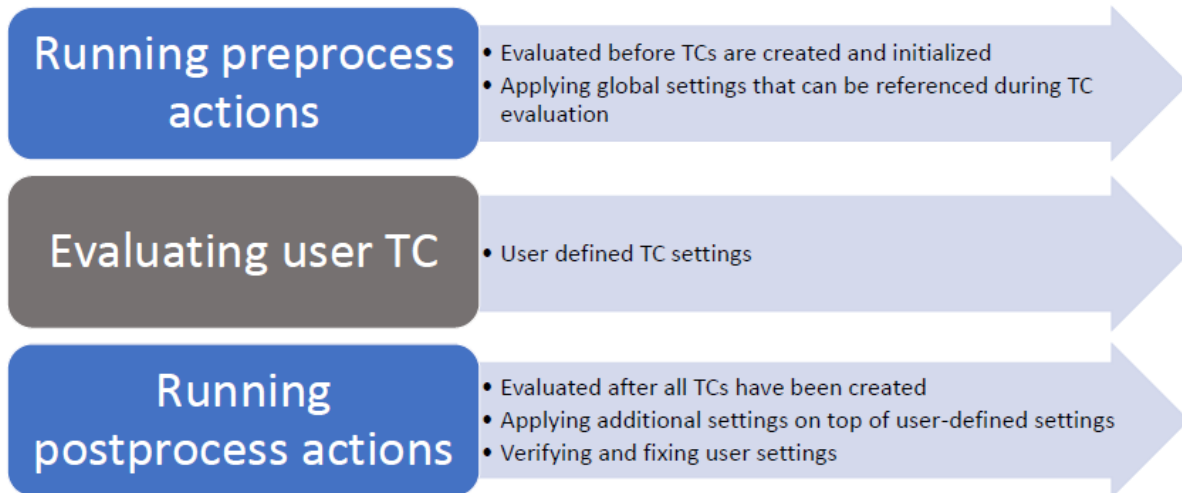
- Create a debug versus a release build of the application
- Add special instrumentation to the application in order to detect run-time errors
- Build the application for different target platforms

[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 or JavaScript expressions 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 parameterize 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.

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

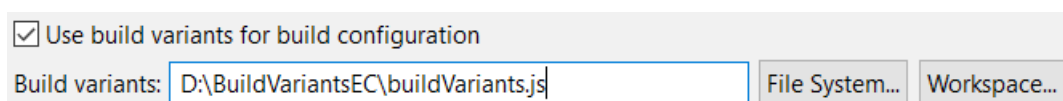
just before ("pre-processing") or just after ("post-processing") the default interpretation of TC properties. Depending on the kind of build variant you want to implement you can choose to write "pre-processing" or "post-processing" scripts (or both). In a pre-processing script you can for example assign 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 (pre-process script, post-process script or both).

To enable the support for build variants, set the preference *RealTime Development – Build/Transformations – C++ – Use build variants for build configuration*. Then set the *Build variants* preference to reference a **build variants script**. This script will be interpreted by Model RealTime 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.

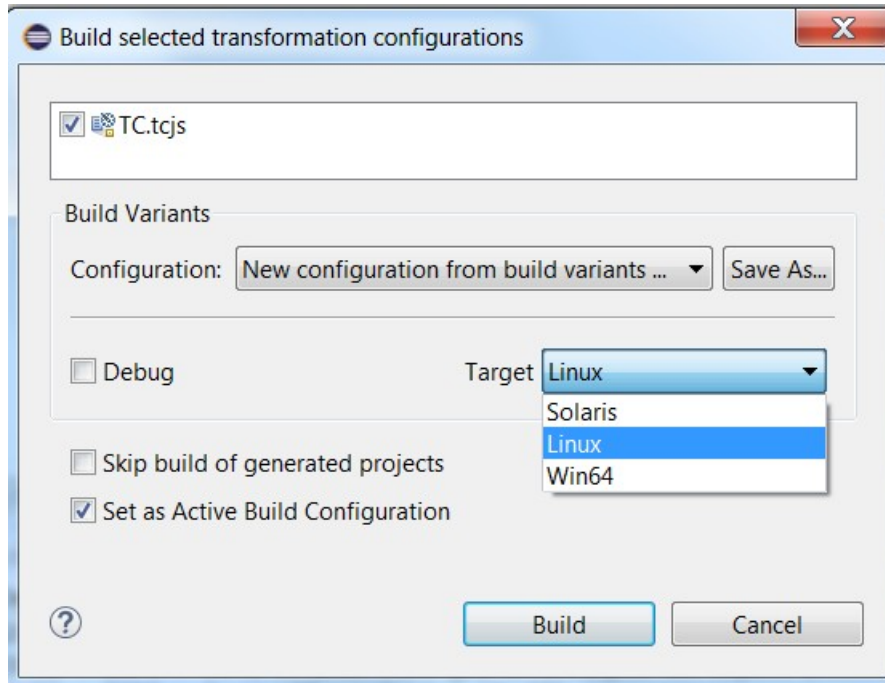
```
let debug = {
  name: 'Debug',
  script: 'debug.js',
  control : { kind: 'checkbox' },
  defaultValue : false,
  description: 'Build for debugging'
};

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 Model RealTime 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.

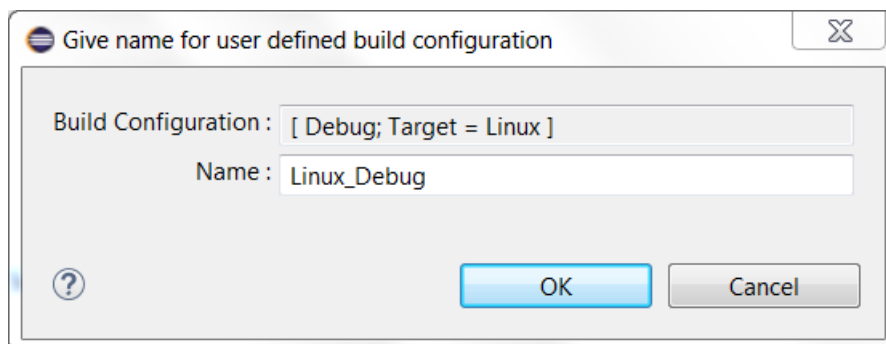
Here is what the user interface will look like when you build a TC with this build variants script enabled:



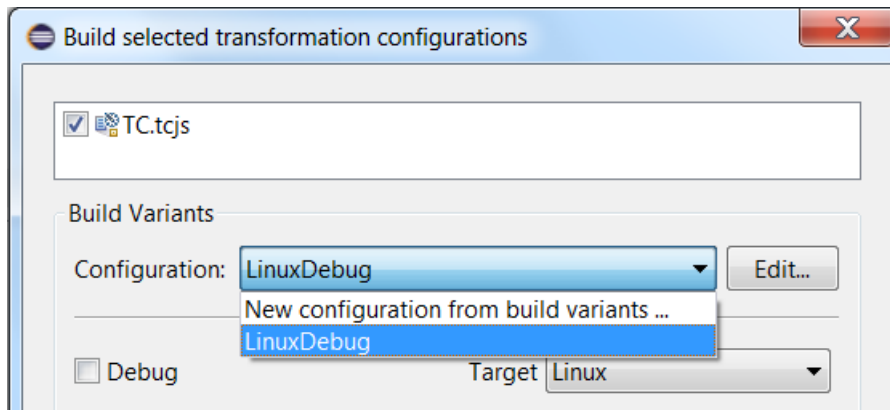
The controls after the separator in the Build Variants group are contributed by the build variants script. When pressing the Build button the TC will first be processed in the usual way. Then the scripts associated with the user interface controls will be invoked so they can modify the TC properties 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 defines 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.

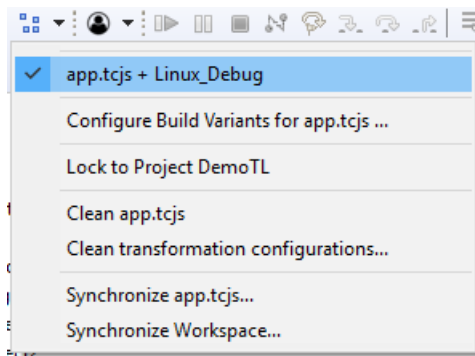
If you need to build a particular build configuration more frequently than others you can save it by pressing the "Save As" button in the dialog. Give the build configuration a descriptive name. For example:



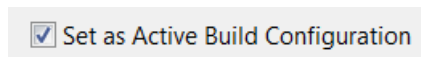
Now you can build this particular build configuration simply by choosing it from the list of build configurations in the dialog.



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 "Configure Build Variants 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 representation 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 at *Model RealTime Java APIs – Model RealTime Transformation Developer's Guide - Reference - API Reference - Transformation Configuration and Build Variants JavaScript API*.

## **Debugging Build Variant Scripts**

If the build variant scripts don't work as you have intended, you need to debug them. In simple cases it may be enough to "debug" by tracing messages to the console. You can for example use the function `BVF.formatInfo()` to print such messages. They will be printed to the Build Variants console if the script runs as part of an interactive build in the UI, and as a model compiler message if the script runs in the context of the model compiler.

Sometimes you may need to do real JavaScript debugging to find a problem in a build variant script. One way to do that is described in the built-in Help at *Model RealTime User's Guide - Articles - Building - Build Variants - Debugging*.

## **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 property is set 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.

**Clean command** specifies a command that cleans the external library. For example it can be an invocation of make clean with a makefile for the library.

**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.

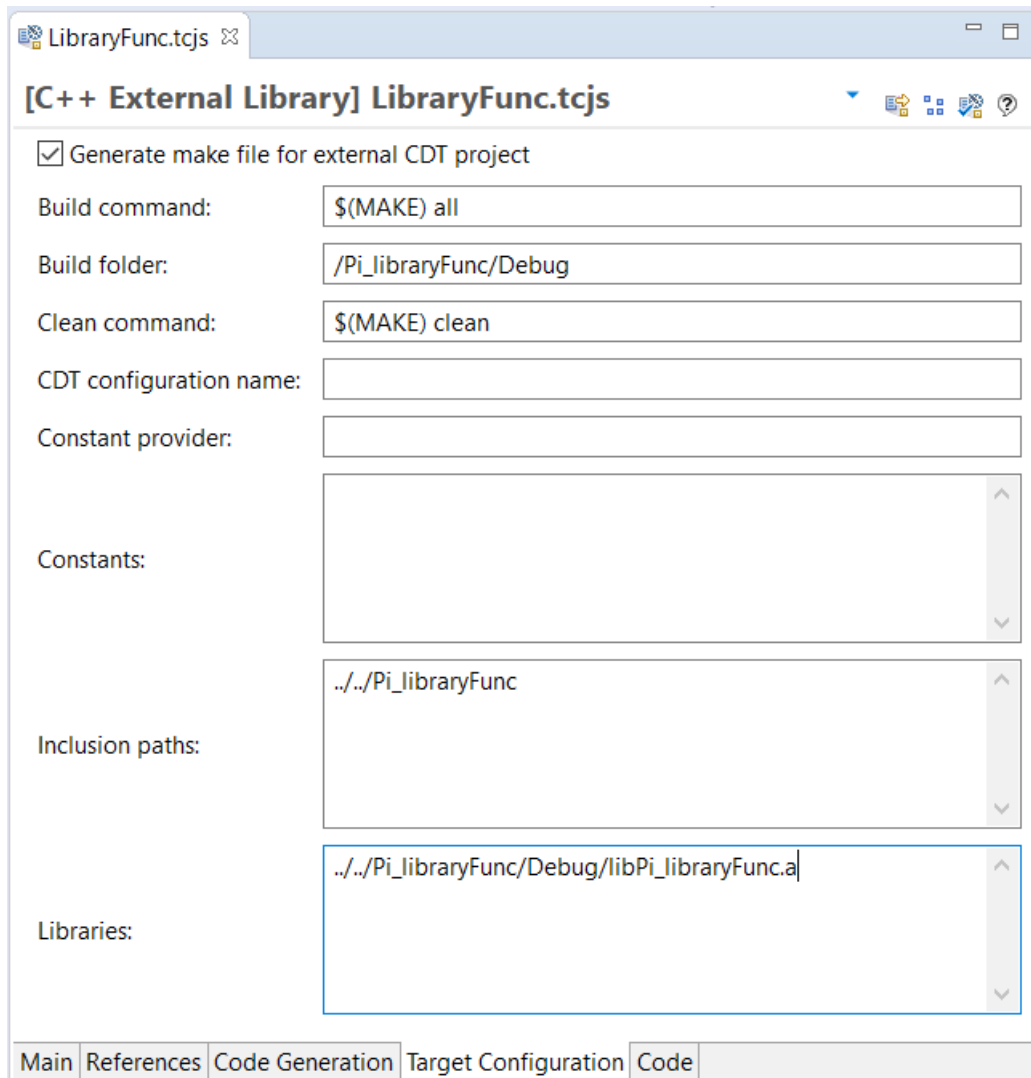
**Inclusion paths** is a list of inclusion paths where the header files of the external library are located. These paths will be added to the include paths in the makefile that is generated for a TC which has the external library TC as a prerequisite.

**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 "Code Generation" 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.

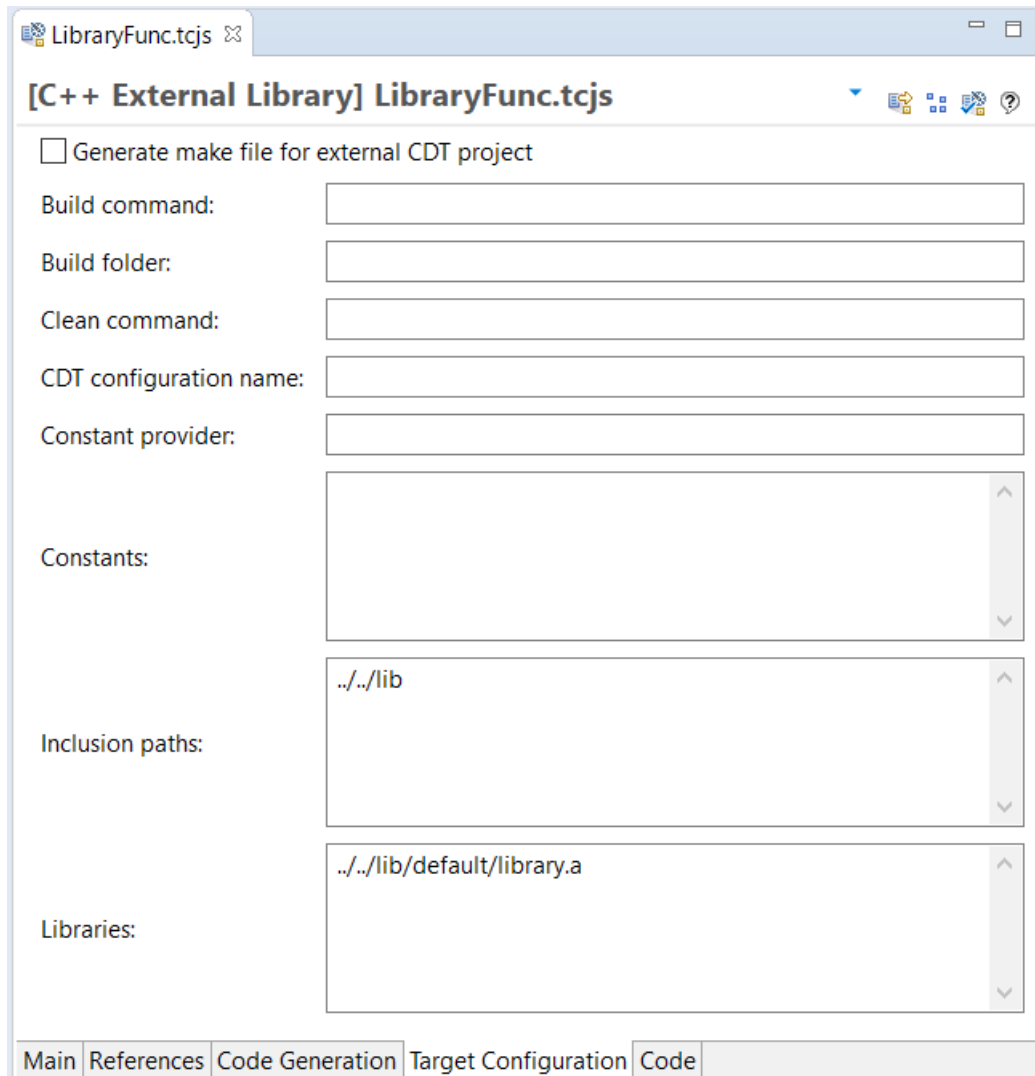


When this TC is built a makefile is first generated for the CDT project Pi\_libraryFunc which should be available in the workspace. The CDT configuration "Debug" will be used, since that target folder is specified in the "Build folder" property. The makefile is processed using the build command "\$(MAKE) 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" properties empty in the "C++ External Library" TC. For example:





When you build an executable TC that has this 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 property **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 model compiler 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
```

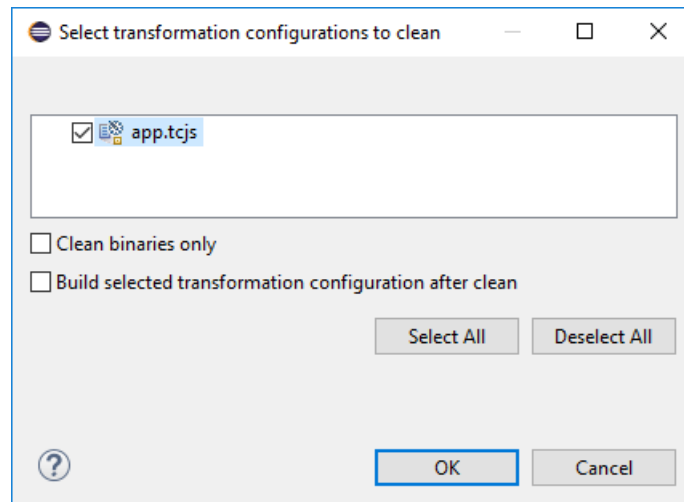
Note that constant values provided by a constant provider plugin override values provided by the Constants property.

## 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.

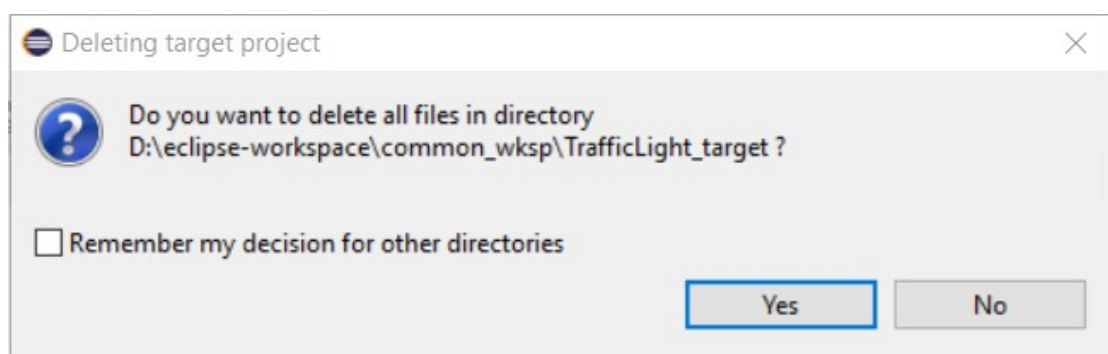
To clean a TC invoke the ”Clean...” command in its context menu. The following dialog will appear:



When you press OK all selected TCs (and their prerequisite TCs, both direct and indirect prerequisites) 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 in order to clean an external project, the "Clean command" property of its TC has to be set. Otherwise Model RealTime 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:

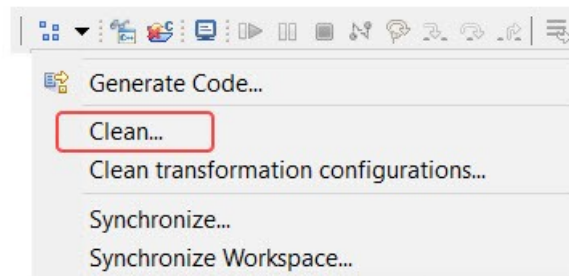


If you answer Yes, the target project folder will be deleted from the file system. If the target project contains subfolders there will popup one such dialog for each subfolder. To apply the same choice to all subfolders (either delete the subfolders or keep them) you can mark the checkbox to remember 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 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 be marked to become cleaned.

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.

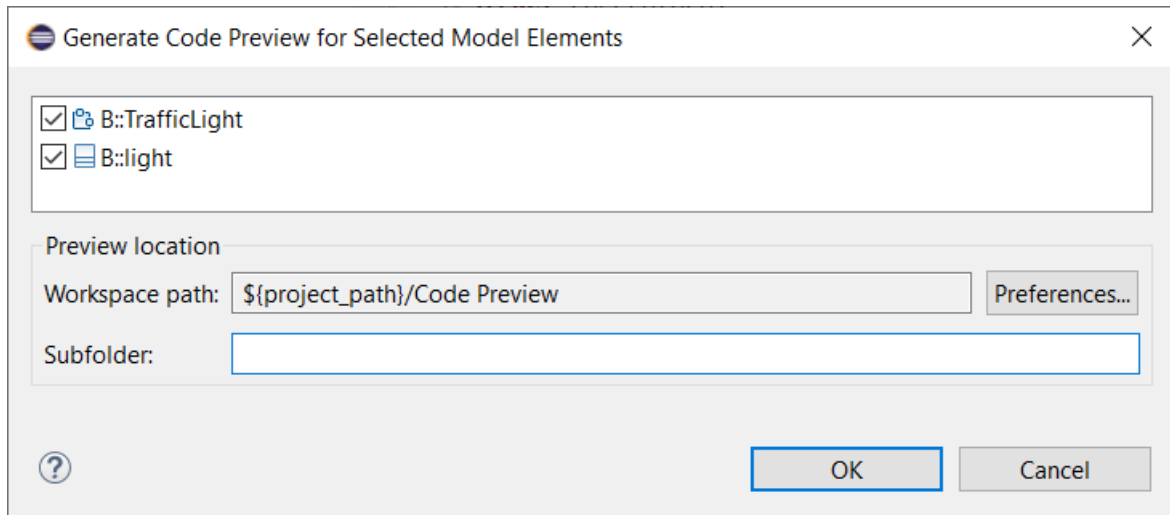
You can quickly clean the whole workspace by using the **Remove All Generated Projects** toolbar button or selecting the command **File - Remove All Generated Projects**. It will remove all generated target projects from the workspace and delete them from the file system.

## Code Preview

Sometimes it can be useful to get a preview of what the generated code will look like, without building a TC in the usual way. For example, you may want to quickly see what the generated code for a certain model element, such as a class, will look like. Such a code preview can help you understand how changes made in the model will affect the generated code.

To generate a code preview for one (or several) model elements use the command **Generate Code Preview** from the toolbar or the Project Explorer context menu. The command has the default keybinding Alt + Shift + E and is available on all model elements that get translated to

their own C++ files, for example a class, capsule or protocol. A dialog appears where you can specify where to place the code preview.



By default the code preview will be placed in a folder called "Code Preview" in the Eclipse project that contains the selected elements. If you have selected elements from multiple projects, the project of the first element will be used. You can change the default code preview location in the preferences. It's also possible to specify a subfolder. This can for example be useful if you want to give a meaningful name to the code preview location that describes what it contains.

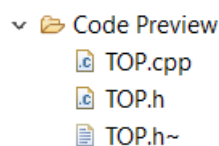
When you press the **OK** button the model compiler will be invoked to generate the code preview for the selected elements. Note that in this case the model compiler does not use a transformation configuration for generating the code. This means you can get a code preview even before you have created your TCs, and it also makes code generation faster. However, there will be some minor differences in a code preview compared to the real generated code because of this. One example is the inclusion of the Unit name header file. Since the name of the unit header file is defined in the TC it will look like this in a code preview:

```
#include <.unitName.h>
```

Usually such differences don't matter since you are not supposed to compare a code preview with the real generated code, but rather with another code preview.

### ***Using Code Preview for Code Comparison***

If you re-generate a code preview into the same location twice, all files that are different from before will be suffixed with the "~" character. Here is an example of what it can look like for a capsule "TOP":



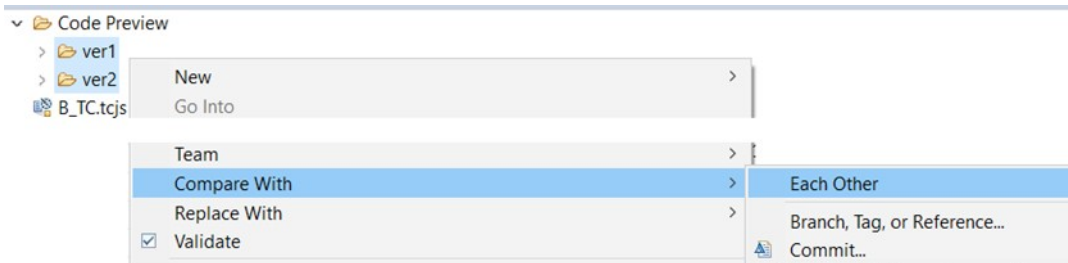
Here we can see that the capsule was changed in a way that affected the generated header file, but not the implementation file.

You can utilize this feature for quickly understanding how some changes you did to a model element will affect the code that is generated for it. Follow these steps:

1. Generate a code preview for the element
2. Make some changes to the element
3. Generate another code preview for the element into the same location
4. Select the new and old version of the generated file (TOP.h and TOP.h~ in the above example), and perform the context menu command **Compare With - Each Other**. In the Compare editor that opens you can easily see how the changes you made to the model element have impacted the generated code.

Another similar scenario is to compare two versions of a model element to see how the generated code for those two versions differs. For example, the versions can be stored on two different branches in Git. Follow these steps:

1. Checkout one of the branches and generate a code preview for the element. Specify the branch name (for example "ver1") as the subfolder name.
2. Checkout the other branch and repeat the same procedure, this time using the other branch name as the subfolder name (for example "ver2").
3. Your "Code Preview" folder will now contain two subfolders "ver1" and "ver2". Select these folders and perform the context menu command **Compare With - Each Other**. The Compare editor will show how the generated code differs between these two versions.

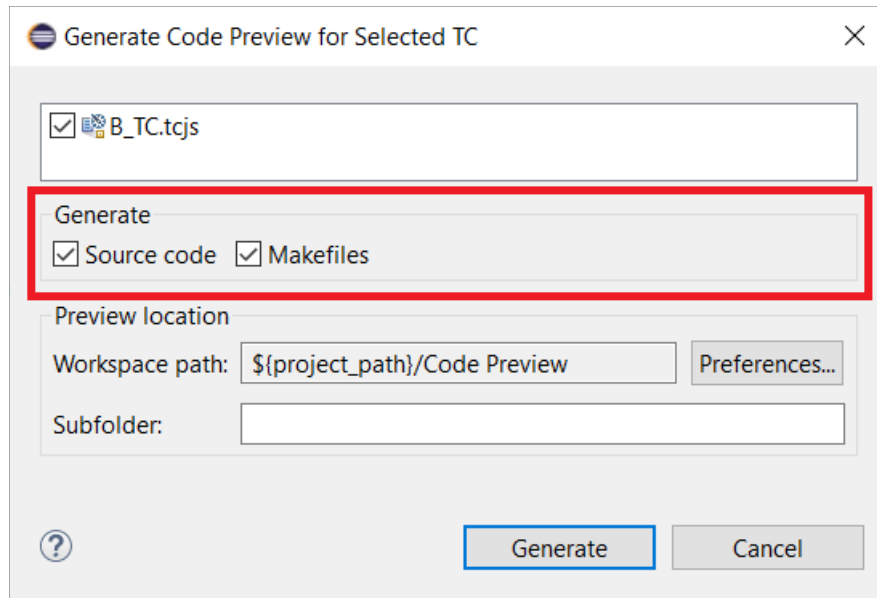


## ***Generating Code Preview for a Transformation Configuration***

The **Generate Code Preview** command is also available in the context menu of a TC. In this case the TC will be used by the model compiler for generating the code preview, which means the code preview will be identical to the code that will be generated when the TC is built.

Generating code preview for a TC is convenient when you want to get a preview of all files that will be generated by that TC when it's built. You can for example use it for comparing one version of a TC and model with another, as explained in [Using Code Preview for Code Comparison](#).

The dialog for generating code preview for a TC has two additional checkboxes which let you choose if you want to also include generated makefiles into the preview:



Comparing the makefiles for two versions of a TC can be a useful tool for troubleshooting build problems.

If the preference for using a [build variant](#) script is set, the above dialog will also contain the user interface provided by the build variant script so that you can set the build variant properties (i.e. build configuration) in the same way as when you build the TC. The values you set for the build variant properties can influence what the generated code will look like.

### ***Removing Code Preview***

When you no longer need a code preview you can simply delete it from the Project Explorer, or delete the folder from the file system and refresh the Project Explorer. There is also a useful command **File - Remove All Code Preview** that will remove all code preview from your workspace. It can be convenient if you have generated lots of code preview and want to remove it all at once.