



DevOps Model RealTime Model Compiler

*Author: Mattias Mohlin
IBM*

Revision history			
0.1	Initial version	Aug 26, 2016	Mattias Mohlin
0.2	Updated for version 10.1 (2017.10) after making the model compiler a non-experimental feature	Mar 14, 2017	Mattias Mohlin
0.3	Updated for version 10.2	Jul 12, 2018	Mattias Mohlin
0.4	Updated for version 10.3	Oct 3, 2018	Mattias Mohlin
0.5	Updated for 10.3 2019.03	Jan 16, 2019	Elena Strabykina
0.6	Updated for 10.3 2019.03	Jan 18, 2019	Mattias Mohlin
0.7	Updated for 10.3 2019.35	Aug 29, 2019	Mattias Mohlin
0.8	Updated for 10.3 2019.43	Oct 29, 2019	Mattias Mohlin
0.9	Updated for 10.3 2020.03	Jan 15, 2020	Mattias Mohlin
1.0	Updated for 11.0 2020.33	Aug 20, 2020	Mattias Mohlin
1.1	Updated for 11.0 2020.39	Sep 28, 2020	Mattias Mohlin
1.2	Updated for 11.0 2021.16	Apr 3, 2021	Alexander Strabykin
1.3	Updated for 11.1 2021.24	June 3, 2021	Mattias Mohlin
1.4	Updated for 11.1.2021.34	August 25, 2021	Vivekanandan Sekaran
1.5	Updated for 11.1 2022.04	February 8, 2022	Mattias Mohlin
1.6	Updated for 11.1 2022.21	May 25, 2022	Mattias Mohlin
1.7	Updated for 11.3 2023.19	May 15, 2023	Mattias Mohlin

Table of Contents

Model RealTime Model Compiler.....	1
Overview.....	3
Location and architecture.....	3
Usage.....	4
Options.....	4
Process Return Value.....	12
Path Maps.....	12
Model Compiler Variables.....	13
Examples.....	16
Build Server.....	17
Model Compiler Console.....	18

This document describes the model compiler in DevOps Model RealTime – what it is and how to use it. All screen shots were captured on the Windows platform.

Overview

In older versions of Model RealTime, generation of code and make files were performed by Eclipse plugins within the Model RealTime development environment. There were some drawbacks with this approach:

- The workspace model was locked for modifications during the time it took to generate code and make files. If the user attempted any action in Model RealTime which required access to the model, the action would block the user until code generation was completed. For big models this could take a few minutes, and it was hard to cancel this operation. This was primarily a usability problem.
- The rules for transforming the model to C++ code had to be relatively straight-forward to ensure a reasonable performance of the code generator. More advanced transformation rules that would require for example preprocessing of the input model, or allocation of non-trivial data structures, were problematic to support since they would impact significantly on the performance and memory consumption of Model RealTime. This problem hence limited what transformation features the code generator could support.
- In batch builds it was necessary to launch Model RealTime in headless mode to perform code generation. This process was fairly time-consuming and also did not work well on systems without display capabilities. This problem caused batch builds to sometimes fail or be too slow.

To overcome these problems Model RealTime now has an improved architecture where generation of code and make files can be performed by a stand-alone utility, called the **model compiler**. This utility is a plain Java program which can run independently of Eclipse. Code generation now takes place outside of the Model RealTime IDE, and the new architecture has solved all of the above mentioned problems.

The model compiler is integrated with the Model RealTime user interface, and although there are some differences compared to the traditional builder (which we now refer to as the classic builder), most things work the same from a user-interface point of view. This means that a user who only runs builds interactively from within Model RealTime does not have to know much about the model compiler. Model RealTime automatically launches it with appropriate arguments when necessary. However, users that need to set-up batch building of Model RealTime models must know how to invoke the model compiler as a stand-alone command-line tool from the build scripts they write. This document describes everything those users need to know about the model compiler.

Location and architecture

The model compiler is a JAR file called `modelcompiler.jar` which can be found in `<install-dir>\plugins\com.ibm.xtools.umltdt.rt.core.tools_<version>\tools`, where `<version>` is a version identifier that depends on the version of Model RealTime. The JAR file depends on a few other JAR files that are located in the subfolder called `modelcompiler_lib`.

If the Eclipse installation is read-only, the installation directory cannot be modified by the Model RealTime installer, and in that case the plugins with the model compiler are written to the writable Eclipse configuration folder. If the Eclipse installation is not read-only, the files will in addition be placed in `<install-dir>\rsa_rt\tools` from where it may be more convenient to access them.

It is recommended to use the same JVM for running the model compiler as is recommended to use for running Model RealTime.

Usage

The model compiler can translate a whole model to C++ and make files in a single invocation. The model compiler can be executed from the command line as follows:

```
java <JVM options> -jar modelcompiler.jar <options> <files>
```

Which JVM options to use depends on the size of the model and capabilities of the machine. Here is an example:

```
-Xverify:none -Xmx4g -XX:+AggressiveOpts -XX:+AggressiveHeap
```

For more information about these, and other available JVM options, please refer to the documentation of your JVM.

At least one of the files that are passed to the model compiler should be a transformation configuration (.tc or .tcjs). It can be specified using a platform resource URI, for example,
platform:/resource/MyProject/my.tc

This gives additional flexibility when defining generic rule to run the model compiler. Platform resource URIs are resolved based on the `--root` option used.

The model compiler builds the specified TC according to the build properties it contains. All model files that are needed for building the TC will be loaded by the model compiler. If you want additional model files to also be loaded you can specify them on the command-line as well.

Options

The following options are available for the model compiler:

Option	Description
<code>--autoComputeAbstractProperty</code>	By default the model compiler will automatically compute if a class or capsule is abstract, by analyzing its local and inherited operations. If the class or capsule contains at least one pure virtual operation that is not redefined or an interface operation that is not implemented, it will be considered abstract. In older versions of the model compiler it was necessary to manually set the Abstract property to be consistent with the operations. Set this option to false if you prefer that behavior.
<code>--build[=<target>]</code>	Build generated code once it has been generated. If this option is not used, the model compiler will only generate C++ code and make files and will not invoke any make tool in order to build the code. This option is typically used when building a model by a

	<p>single invocation of the model compiler from the command-line. The make command to be used is taken from the TC. By default the make target “all” will be used, but you can specify a different make target after an equal sign. For example:</p> <pre>--build=mytarget</pre>
<code>--buildConfig="config"</code>	<p>Specify a concrete build configuration for the current build. A build configuration is a string holding a semicolon-separated list of exact settings for build variants declared in the build variants file. Each such setting maps to invocations of build variant scripts being invoked during the build. See the documentation about Build Variants for more information.</p> <p>For example:</p> <pre>--buildConfig="Platform=Linux64;Config=Debug;GNU Cov"</pre>
<code>--buildVariants=<file></code>	<p>Specify a JavaScript file that initializes build variants available for builds. See the documentation about Build Variants for more information.</p>
<code>--codeStandard=<arg></code>	<p>By default the model compiler generates code which requires a C++ 17 compiler, but if you use a compiler supporting a different language standard you can set this option. The following arguments are accepted:</p> <ul style="list-style-type: none"> • c++11 Generate code which requires a C++ 11 compiler. • c++14 Generate code which requires a C++ 14 compiler. • c++17 Generate code which requires a C++ 17 compiler. This is the default. • c++20 Generate code which requires a C++ 20 compiler. • pre-c++11 Generate code for an older C++ standard (before C++ 11). Note that for this code standard you also need to use an older version of the TargetRTS which doesn't contain any C++ 11 constructs. This option can hence be useful if you want to use the latest version of Model RealTime to develop applications that were originally created with older versions of Model RealTime where a C++ 11 compiler was not used.
<code>--codeCompliance=<arg></code>	<p>Use this option to ensure that the generated code complies with certain rules checked by static code analysis tools.</p>

	<p>Currently the following arguments can be used (corresponding to supported static code analysis tools):</p> <ul style="list-style-type: none"> • clang-tidy Generated code will be compliant with certain Clang-Tidy checks. <p>Depending on situation the generated code will be made compliant either by being adapted to avoid constructs that static code analysis tools complain about, or by inserting special comments into the code to suppress checks for code constructs that are known to be correct.</p>
<code>--cwd=<path></code>	Sets the current working directory. If you do this you can then use paths relative to that working directory, for example when specifying the TC file, or the root folder.
<code>--env=<file></code>	<p>Specifies an environment file, which is a text file containing variables that control the detailed behavior of the model compiler. Each variable should be specified on a separate line in the file according to the format</p> <pre><variable-name>=<variable-value></pre> <p>It is also possible to use command-line options, environment variables or JVM system properties for specifying values for these variables.</p> <p>For more information and a list of all available variables that can be used see Model Compiler Variables.</p>
<code>--exportModelMapping=<file></code>	This option generates a file with information about how model elements are mapped to places in generated code. This information is used by Model RealTime when it launches the model compiler to implement various navigation commands. You should not use this option yourself when calling the model compiler.
<code>--exportMsg=<path></code>	Causes the model compiler to generate a file (in the specified folder) containing messages produced during code generation. These messages are also printed to the console, but with less detail. The generated file is primarily used when integrating the model compiler with the Model RealTime user interface, and you will normally not use this option otherwise.
<code>--forceOverwrite</code>	By default the model compiler will not regenerate a file if its contents have not changed from last time it was generated. This ensures that the make tool will not build source files that have not been changed. If you set this option the model

	<p>compiler will overwrite all files even if their contents have not changed.</p>
<code>--genBuildInfoRules</code>	<p>This option can be used to generate additional rules into the make file which will print a file with information about the build. The file is in JavaScript syntax and contains information about, for example, which source elements are built, which compiler that is used, the name of the target binary etc.</p> <p>Note that this option only works for inclusive make files, so ensure that you also specify the <code>--inclusiveBatch</code> option.</p> <p>To generate the information file run this command after running the model compiler command: <code><make> -f batch.mk build_info</code></p> <p>A file called <code>TC_build_info.js</code> (where TC is the name of the built TC) will be generated.</p>
<code>--genSource=</code> <code>"<elements>"</code>	<p>This option can be used to restrict the model compiler to only generate code for a specific set of model elements. You can specify the elements in a few different ways:</p> <ul style="list-style-type: none"> • By the XMI id of the element. In this case prefix the id with '#'. • By the fully qualified name of the element. In this case prefix the qualified name with '@'. • By the platform resource URI of the element. This URI has the form <code>platform:/resource/<project>/<file>#<xmi id></code> <p>Separate the elements with spaces in the argument string.</p> <p>Example: <code>--genSource="#_S-PlsGoLEeaMRsIRlk4stQ @MyPkg::MyClass platform:/resource/NewCGTest/HelloWorld.emx#_SHU8AGX-EeaTEcy2sB6amw"</code></p> <p>Generate source code for three elements (the first one is specified by its XMI id, the second by its fully qualified name and the third by its platform resource URI).</p>
<code>--genUnit=<file></code>	<p>Tells the model compiler to generate the unit header and implementation files for the argument TC file. No other source code files will be generated, unless the <code>--genSource</code> option is also used.</p>
<code>--generate=<arg></code>	<p>Specifies what type of files should be generated by the model compiler. The following arguments are supported for this option:</p> <ul style="list-style-type: none"> • <code>all</code> Both make files and source code files are generated.

	<p>This is the default.</p> <ul style="list-style-type: none"> • makefile Make files are generated. • source Source code files are generated. A CDT project file is also generated. You can import it into an Eclipse workspace for looking at the generated source code files.
--help	Prints all available options with brief descriptions to the console. The model compiler then terminates.
--inclusiveBatch	This option corresponds to the model compiler variable <code>RTMakeMode=inclusive</code> (see RTMakeMode).
--keepBuilding	If multiple TC files are passed to the model compiler its default behavior is to build them one by one. If building one of the TCs fails remaining TCs will not be built. Set this option to change this behavior so that all TCs are built regardless if errors occur.
--license=<arg>	<p>Specifies where to obtain a license for the model compiler.</p> <p>The format of the argument depends on the kind of license you have.</p> <p>It is on the form <code><type>:<key></code></p> <p>The <code><type></code> should either be “flex” (floating license) or “key” (authorized user license).</p> <p>The <code><key></code> specifies the actual license. For “flex” it should be a license server host address: <code><port>@<hostaddress></code></p> <p>For “key” it should be the license key string.</p> <p>Not all features of the model compiler require a license. If you get a message about a missing license, you need to use the <code>--license</code> option to specify the license to use. In case of a floating license the model compiler keeps the license checked out until it terminates.</p>
--list=<arg>	Use this option if you want information printed about what the model compiler will generate. By default information is printed for all TCs that are built. This information includes for example the type of TC (executable, library etc.), the

	<p>path to the target folder where generated files will be placed, etc.</p> <p>If you set <code><arg></code> to “sources”, information will instead be printed about all source model elements that will be transformed. You will see the fully qualified name and kind of each source element.</p>
<code>--optionsfile=<file></code>	<p>Reads model compiler options from a file instead of specifying them on the command-line. This may be useful in case you run into too long command-lines or simply just prefer to edit options in a text editor.</p> <p>The options file should be a text file with each option specified on a separate line according to the format</p> <pre><option-name>=<option-value></pre> <p>For options that do not take a value, the format is</p> <pre><option-name>=</pre> <p>Here is an example:</p> <pre>root=C:\myworkspace env=C:\modelcompiler\mcenv.map out=C:\modelcompiler build=</pre> <p>If an option is specified both on the command-line and in an options file, then its value on the command-line takes precedence.</p>
<code>--organizeSources</code>	<p>This command corresponds to the Organize Sources button on the Main tab of the TC Editor. The model compiler will analyze source dependencies based on model references. For an executable TC the analysis starts from the specified top capsule, while for other TCs it starts from the specified source elements. The result of the analysis is a list of elements that are suggested to be added as new source elements of the TC, and a list of elements that are suggested to be removed from the sources list. The messages are printed on a format so you can directly use text from them to update the "sources" TC property using the Code tab in the TC editor.</p>
<code>--out=<path></code>	<p>Specifies the output folder for the model compiler. The workspace output path that is specified in the TC will be appended to this path to define the location where all generated files will be placed.</p>
<code>--pathmap=<path></code>	<p>Specifies a file with path maps. This option must be used if</p>

	<p>your model contains pathmap URIs (URIs that start with “pathmap”, for example <code>href="pathmap://RT_SAMPLE_LIB/RTSampleCustomLibrary.emx#_FPfjMF83EeiD6r1CftWc8g?RTSampleCustomLibrary/BaseA?"</code>).</p> <p>See Path Maps for more information.</p>
<p>--root=<path> or --root=<map file></p>	<p>Specifies the root folder for the model compiler. All references in models and TC files will be resolved based on this root folder. It is therefore usually set to the workspace folder:</p> <p>--root=<path to existing workspace></p> <p>In this case the model compiler will automatically detect the location of all projects based on workspace metadata (no matter if they are imported into the workspace from other locations or not).</p> <p>The <path> argument can use constructions like "<path to dir>*" or "<path to dir>/*" to enable auto-scan for all sub-folders when resolving platform resource URIs. Note that path must be quoted in this case.</p> <p>It is also possible to specify individual locations of all projects in a map file and provide it as an argument for this option. A map file is a text file where the location of each project is written on a separate line, like this:</p> <p><project-name>=<path-to-project-folder></p> <p>Environment variables and system properties can be used inside the map file using the syntax \$(var) or \${var}.</p> <p>The --root option can be specified multiple times. All values are merged. The last value provided overrides previous ones.</p>
--rteSharedLoc=<path>	<p>This option specifies the location where Model RealTime is installed. The model compiler may need this information in case the built model references something that is located in the Model RealTime installation.</p>
--ruleConfiguration=<rules>	<p>This option can be used for configuring which validation rules the model compiler should check the input model against. In case a rule fails and a problem will be reported, the option can also specify which severity the problem should have (Information, Warning or Error). The <rules> is a comma-separated list of rule ids prefixed with with one of the following letters:</p> <ul style="list-style-type: none"> • X: Disable the validation rule (so it will not be used when checking the model)

	<ul style="list-style-type: none"> • E: Set the severity of the validation rule to Error • W: Set the severity of the validation rule to Warning • I: Set the severity of the validation rule to Information <p>Here is an example: X0001,W0002,E0003 (disable rule 0001, enable rule 0002 and set its severity to Warning, and enable rule 0003 and set its severity to Error).</p> <p>You should only configure a rule once, but if you do it multiple times, the last configuration will take precedence. The id of a rule can be seen from the message that gets printed when the rule is enabled and fails. The id will be printed right after the severity in the build message.</p>
--syncMap=<path>	This option is used when the model compiler is launched by Model RealTime in order to request it to print certain information that is needed for code-to-model synchronization. You don't need to use this option yourself when calling the model compiler.
--timing=<file>	Prints information to a file about the time it took to run the model compiler. This option may be useful if you want to test so that build times remain acceptable when making changes to your models or the build system.
--validate=<rule>	<p>Sets the validation rules to be performed by the model compiler. The following rules are supported:</p> <ul style="list-style-type: none"> • sources Validate source model elements and report errors if any source element is missing • tc TC files are validated. <p>You can set multiple rules separated by a comma. Set the scope to off in order to disable all validation. By default all validations are turned on.</p>
--verbose=<level>	<p>Sets the verbosity of the model compiler. The following verbosity levels are supported:</p> <ul style="list-style-type: none"> • default By default the model compiler is rather verbose and prints several information messages to the console to indicate its progress. This level is appropriate when running the model compiler as a stand-alone tool from the command-line. • makefile Disables all console printouts. This level is typically appropriate when the model compiler is invoked from a make file, which may have its own printouts.

<code>--version</code>	Prints the version of the model compiler.
------------------------	---

Process Return Value

The model compiler exits with a non-zero return value in case the build fails. In that case there will also be a printout with an explanation of why it failed.

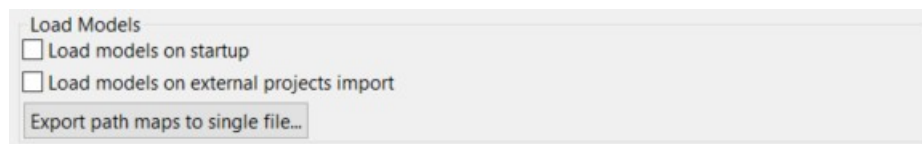
Path Maps

Path maps are variables that allow URIs to be more portable. A URI can contain a path map variable that can be resolved to different values in different environments. Here is an example of a model reference that uses a URI that contains a path map variable “RT_SAMPLE_LIB”:

```
href="pathmap://RT_SAMPLE_LIB/
RTSampleCustomLibrary.emx#_FPfjMF83EeiD6rlCftWc8g?RTSampleCustomLibrary/
BaseA?"
```

Path map variables can be defined in the Model RealTime preferences (*Modeling – Path maps*).

The model compiler needs access to the values of all path map variables that are used within the built model. The command-line option `--pathmap` should be used to specify a file that contains the path map variables and their values. The easiest way to get such a file is to generate it from inside Model RealTime. There is a button on the *RealTime Development* preference page that allows you to do this.



The file is on the following form:

```
[map]
<variable>
...
<variable>
```

Each `<variable>` specifies a path map variable in one of the following ways:

Note: NAME is the path variable name, and all paths should use ‘/’ as separator (also on Windows).

NAME=file:<path to folder>

For a file located in a folder

NAME=jar:file:<path to jar file>!<path to folder within jar file>

For a file located inside a JAR file.

pathmap://NAME/<file name>**=file:**<path to file>

This syntax allows you to map a certain pathmap URI to a specific file located in a folder.

pathmap://NAME/<file name>**=jar:file:**<path to jar file>!<path to file within jar file>

This syntax allows you to map a certain pathmap URI to a specific file located in a JAR file.

It is possible to use environment variables within path map variable definitions. Use the syntax `{VAR}` or `$(VAR)`. Environment variables are substituted first before parsing the path map variable definition.

You can use comments in the file. Write them on a line that starts with “#”.

Here are some examples of path map variable definitions:

Mapping of path map variable to folder in jar archive [map] RT_SAMPLE_LIB = jar:file:/D:/work/tmp/plugins/sample.jar!/libraries
Direct mapping of pathmap URI to physical file [map] pathmap://RT_SAMPLE_LIB/RTSampleCustomLibrary.emx=file:/D:/tmp/sample.emx
Direct mapping of pathmap URI to file within jar archive [map] pathmap://RT_SAMPLE_LIB/RTSampleCustomLibrary.emx = jar:file:/D:/tmp/pathmap.jar!/entry2.emx
Mapping of path map variable to folder in the file system [map] RT_SAMPLE_LIB = file:/D:/work/tests/com.hcl.test.profiles.and.libs/libraries

Model Compiler Variables

The detailed behavior of the model compiler is controlled by a number of variables. These correspond to those preferences in Model RealTime which affect the result of building a model. Variables can be specified in an environment file and passed to the model compiler using its `--env` command-line option. They can also be specified as environment variables or as JVM system properties on the command-line. The model compiler also provides dedicated command-line options for setting the variables. If the same variable is set using several of these mechanisms the variable’s value is obtained in this priority order (from high to low):

1. Variable defined in environment file
2. JVM system property
3. Environment variable

When you call the model compiler from the command-line (or from a script) you will normally use command-line options for setting variables. The use of an environment file is mainly intended for the integration between the Model RealTime Eclipse IDE and the model compiler.

The following variables can be used:

Variable	Type	Description
RTAutoDeps	Boolean	By default, the C++ transform uses the Sources list of the transformation configuration to determine which model elements to transform to C++. It is therefore important to maintain this list to keep it correct and minimal at all times. If you set this preference the Model Compiler will analyze dependencies based on the top-level capsule (all elements it depends on, directly and indirectly), and add missing sources for the time of the build without modifying the source list in the TC file. All added

		sources will be printed into the UML Development Console. This variable corresponds to the preference <i>RealTime Development – Build/Transformations – C++ – Detect Source Dependencies Automatically</i> .
RTAutoDepsLog	Boolean	This variable applies if RTAutoDeps or RTContextSensitive is set to true. The model compiler will then print additional logging about which dependent elements that are automatically included in the build. This variable corresponds to the preference <i>RealTime Development – Build/Transformations – C++ – Report details about automatically added source elements</i> .
RTCDRUNVariable	String	Specifies the location of the cdrun.pl script. By default the cdrun.pl from the Model RealTime installation is used.
RTCodanSupport	Boolean	If set to true, the generated CDT project will be configured to use static code analysis (using CDT Codan). This variable corresponds to the preference <i>RealTime Development – Build/Transformations – C++ – Generate additional information for Code Analysis</i> .
RTComplexTypeDescriptors	Boolean	If set to true, the model compiler will attempt to generate type descriptors also for complex types. This variable corresponds to the preference <i>RealTime Development – Build/Transformations – C++ – Generate type descriptors for complex types</i> .
RTContextSensitive	Boolean	If set to true, the model compiler will build minimal versions of prerequisite libraries based on the context in which they are used. This means that only those parts of libraries that are needed for the current build will be generated and compiled which can reduce the build time significantly. This variable corresponds to the preference <i>RealTime Development – Build/Transformations – C++ – Context sensitive library builds</i> .
RTDependVariable	String	Specifies the location of the rtcppdep.pl script. By default the rtcppdep.pl from the Model RealTime installation is used.
RTDetectTransitionCycles	Boolean	If set to true, the model compiler will analyze state machine transition graphs when building a TC. Warnings are reported if cycles in the transition graph are found. Such cycles may lead to infinite loops when executing the generated application. This variable corresponds to the preference

		<i>RealTime Development – Build/Transformations – C++ – Check transition cycles</i>
RTLineSeparator	String	Specifies which kind of newlines to use in generated files. This variable corresponds to the preference <i>General – Workspace – New text file line delimiter</i> .
RTLinkOrder	String	Specifies the default link order for libraries when building an executable. This variable corresponds to the preference <i>RealTime Development – Build/Transformations – C++ – Link order</i> .
RTMakeMode	String	<p>Controls the kind of make file that is generated. It can be either “recursive” (default) or “inclusive”. This variable corresponds to the preference <i>RealTime Development – Build/Transformations – Type of Generated Make Files</i>.</p> <p>By default the make files that are generated by the model compiler are recursive, meaning that make is called recursively from the make files in order to build prerequisite libraries. By inclusive make file we mean a make file that includes other make files. Use of inclusive make files mean that a single invocation of make is enough for building everything. Make tools that support parallel processing of make rules often work more efficiently if inclusive make files are used.</p>
RTMissingOutgoingTransitions	Boolean	<p>If set to true, warnings will be reported if pseudo states with missing outgoing transitions are found. This variable corresponds to the preference</p> <p><i>RealTime Development – Build/Transformations – C++ – Detect missing outgoing transitions</i></p>
RTOptimizeMArrays	Boolean	<p>If set to true, the code for initialization of multi-dimensional arrays will be optimized to use a single for-loop. This variable corresponds to the preference <i>RealTime Development – Build/Transformations – C++ – Optimize initialization code for multi-dimensional arrays</i>.</p>
RTPerlVariable	String	Specifies the location of the rtp Perl executable. By default the rtp Perl from the Model RealTime installation is used.
RTPropertySetsDefaultsModel	String	This variable can be used to specify a model file that contains custom default values for Property Sets properties (those properties that you see on the "C++ General" and "C++ Target RTS" property pages). If specified it overrides any such model file specified in the built model itself. The variable hence corresponds to the

		preference <i>Modeling – Profiles – Property Sets – Defaults</i> .
RTReportWarningsAsErrors	Boolean	If set to true, all warnings produced by the Model Compiler will be reported as errors. This variable corresponds to the preference <i>RealTime Development – Build/Transformations – C++ – Report warnings as errors</i>
RTSetupVariable	String	Specifies the location of the rtsetup.pl script. By default the rtsetup.pl from the Model RealTime installation is used.
RTToolsVariable	String	Specifies the location of the tools folder. By default the tools folder from the Model RealTime installation is used.
RTValidateMissingSources	Boolean	By default, the model compiler uses the Sources list of the transformation configuration (TC) to determine which model elements to transform to C++. It is therefore important to maintain this list to keep it correct. This validation rule will analyze the top-level capsule, and all elements it depends on (directly and indirectly), and report errors if some referenced elements are missing in the Sources list.

Examples

Below are some examples of calling the model compiler from the command-line.

```
java -jar modelcompiler.jar --help
```

Print information about the options supported by the model compiler.

```
java -jar modelcompiler.jar --root=C:\myworkspace --out C:\outputdir --build C:\workspace\project\HelloWorld.tcjs
```

Generate source code and make files for a TC file. Then build generated code by running make.

```
java -jar modelcompiler.jar --root=C:\paths.map --out C:\outputdir --build C:\workspace\project\HelloWorld.tcjs
```

As above, but instead of specifying the workspace as root folder, use a map file which defines the location of the workspace projects in the file system.


```
java -DRTMakeMode=inclusive -jar modelcompiler.jar --root=C:\paths.map --out C:\outputdir --build C:\workspace\project\HelloWorld.tcjs
```

As above, but generate an inclusive instead of recursive make file.

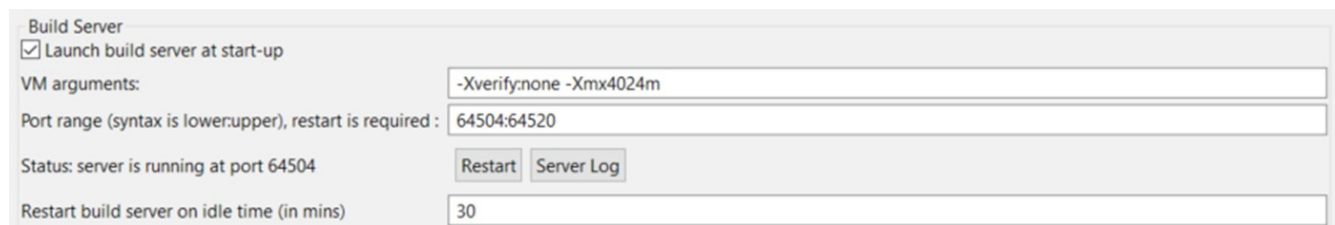
```
java -jar modelcompiler.jar --root=C:\myworkspace --out C:\outputdir -generate=makefile C:\workspace\project\HelloWorld.tcjs
```

Only generate make files for a TC file.

Build Server

The model compiler can be run in a special mode where it acts as a build server. Model RealTime uses the build server to improve performance by not having to start many instances of the model compiler, that only will run for a short while. The model compiler is started as a server once for each running instance of Model RealTime. This process is fully automatic and not something you need to know much about. It is not possible or meaningful to manually start the model compiler in server mode.

The preferences in *RealTime Development – Build/Transformations – Build Server* control how to start the build server. You need to ensure that the specified port range is big enough so that each instance of Model RealTime that you will run can have its own build server running. Press the Restart button on this preference page to restart the build server or start it if you find that it (for whatever reason) is not running as expected. There is also a button for opening the build server log, which can help you troubleshoot any problem you suspect may be related to the build server.

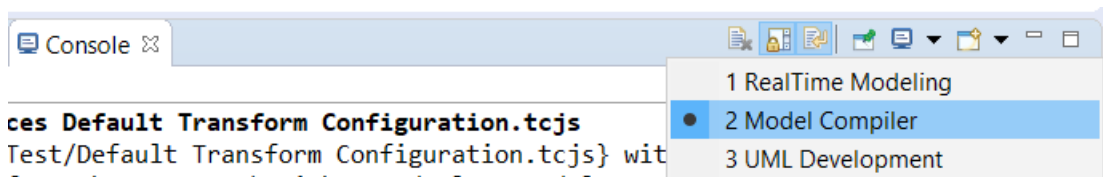


It is possible to prevent the build server from being launched at start-up. Only do this if you will not perform any code generation or work with transformation configurations.

Set the preference "Restart build server on idle time" to let the build server automatically restart when it has been idle for a certain number of minutes. This can reduce the memory consumption of the build server.

Model Compiler Console

Model RealTime provides a special console where messages from the build server are printed. This console is called "Model Compiler" and you find it in the Eclipse Console view.



Usually you don't need to look in this console, but in case a command fails that you suspect could be related to the build server, you can look here for information (or in the Build Server log mentioned above).

One scenario when the model compiler console is very useful is when you want to know how a certain command that you performed from the user interface should be invoked from the command-line. Look for an information message on this form:

INFO : Use this command line to perform the same task with stand-alone Model Compiler:

The command is printed so that you can just copy and paste it on the command-line to use it directly.