

RSARTE Model Compiler

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

Table of Contents

Overview....2
Location and architecture....2
Usage....2
Options....3
Process Return Value....8
Path Maps....8
Model Compiler Variables....9
Examples....11

This document describes the model compiler in Rational Software Architect Real-Time Edition (RSARTE) – what it is and how to use it. All screen shots were captured on the Windows platform.

Overview

In older versions of RSARTE, generation of code and make files were performed by Eclipse plugins within the RSARTE 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 RSARTE 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 RSARTE. This problem hence limited what transformation features the code generator could support.
- In batch builds it was necessary to launch RSARTE 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 RSARTE 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 RSARTE IDE, and multiple invocations of the model compiler can run in parallel in order to build a big model very quickly. The new architecture has solved all of the above mentioned problems.

The model compiler is integrated with the RSARTE user interface, and although there are some differences compared to the traditional builder (which we 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 RSARTE does not have to know much about the model compiler. RSARTE automatically launches it with appropriate arguments when necessary. However, users that need to set-up batch building of RSARTE 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.umldt.rt.core.tools_<version>\tools, where <version> is a version identifier that depends on the version of RSARTE. The JAR file depends on a few other JAR files that are located in the subfolder called `modelcompiler_lib`.
It is recommended to use the Oracle JVM 8 or later for running the model compiler (same as is recommended to use for running RSARTE).

Usage

The model compiler can translate a whole model to C++ and make files in a single invocation. However, it can also be used for generating only a few C++ files from a small part of the model. This

capability makes it possible to use it as a tool in a make file which can drive all steps of building an RSARTE model into executables and libraries. A make tool that supports parallel execution can hence process such a make file very efficiently, which ensures optimal overall build performance.

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 TC/TCJS file. It can be specified using platform resource URI, for example, `platform:/resource/MyProject/my.tc` This gives additional flexibility when defining generic rule to run model compiler. Platform resource URI is resolved based on `--root` setting from options.

Model compiler builds the specified TC according to the build settings 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
`--build[=<target>]`	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 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: `--build=mytarget`
`--buildVariants=<file>`	Specify JavaScript file that initializes build variants available for builds.
`--buildConfig=”config”`	Specify concrete build configuration for current build. Build configuration is a string holding a semicolon-separated list of exact settings for build variants declared in the build variants file.

	For example: --buildConfig=”Platform = Linux64; Config = Debug; GNU Cov”
`--cwd=<path>`	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.
`--env=<file>`	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 `<variable-name>=<variable-value>` It is also possible to use command-line options, environment variables or JVM system properties for specifying values for these variables. For more information and a list of all available variables that can be used see Model Compiler Variables.
`--exportModelMapping=<file>`	This option generates a file with information about how model elements are mapped to places in generated code. This information is used by RSARTE when it launches the model compiler to implement various navigation commands. You should not use this option yourself when calling the model compiler.
`--exportMsg=<path>`	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 RSARTE user interface, and you will normally not use this option otherwise.
`--forceOverwrite`	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 compiler will overwrite all files even if their contents have not changed.
`--genBuildInfoRules`	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. Note that this option only works for inclusive make files, so ensure that you also specify the `--inclusiveBatch` option. To generate the information file run this command after running the model compiler command: `<make> -f batch.mk build_info`

	A file called TC_build_info.js (where TC is the name of the built TC) will be generated.
`--genModelRules=<arg>`	By default the model compiler generates make files with rules for how to compile and link the generated source code files. That is, the model compiler first generates all source code files and make files, and then invokes a make tool on the generated make file for compiling and linking. If you use the `—genModelRules` option additional make rules will be inserted in the make file. These rules specify how to transform model files into source code files by launching the model compiler. Hence, this makes it possible to control the entire build process from a single make file. A make tool that supports processing of multiple make rules in parallel can therefore potentially perform a full build, including both source code generation and compilation, very fast. The argument to the `—genModelRules` option can be one of the following: • `separate` One make rule will be generated for each source file (or pair of C++ implementation and header file). This is the default and allows for maximum parallelization of the make step. • `joint` One make rule will be generated for all source files that correspond to the model elements stored in the same model file. This typically leads to fewer invocations of the model compiler when processing the make file, but also means less possibility to parallelize the make step. • `full` Only one make rule will be generated for all source files. This leads to a single invocation of the model compiler and can be useful if your make tool does not support parallel execution of make rules, but you still want to drive the complete build process from a single make file.
`--genSource= "<elements>"`	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: • 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 `platform:/resource/<project>/<file>#<xmi id>` Separate the elements with spaces in the argument string.

	Example: `--genSource="#_S-PlsGoLEeaMRsIRlk4stQ @MyPkg::MyClass platform:/resource/NewCGTest/HelloWorld.emx#_SHU8AGX-EeaTEcy2sB6amw"` 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).
`--genUnit=<file>`	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 `—genSource` option is also used.
`--generate=<arg>`	Specifies what type of files should be generated by the model compiler. The following arguments are supported for this option: • `all` Both make files and source code files are generated. This is the default. • `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 `RTMakeMode=inclusive` (see RTMakeMode).
`--license=<arg>`	Specifies where to obtain a license for the model compiler. The format of the argument depends on the kind of license you have. It is on the form `<type>::<key>` The `<type>` should either be “flex” (floating license) or “key” (authorized user license). The `<key>` specifies the actual license. For “flex” it should be a license server host address: `<port>@<hostaddress>` For “key” it should be the license key string. Not all features of the model compiler require a license. If you get a message about a missing license, you need to use the `--license` option to specify the license to use. In case of a floating license the model compiler keeps the license checked out until it terminates.
`--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 path to the target folder where generated files will be placed, etc. If you set `<arg>` 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.

`--optionsfile=<file>`	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.
	The options file should be a text file with each option specified on a separate line according to the format
	
	`<option-name>=<option-value>`
	
	For options that do not take a value, the format is
	
	`<option-name>=`
	
	Here is an example:
	
	`root=C:\myworkspace`
	`env=C:\modelcompiler\mcenv.map`
	`out=C:\modelcompiler`
	`build=`
	
	If an option is specified both on the command-line and in an options file, then its value on the command-line takes precedence.

`--out=<path>`	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.
`--pathmap=<path>`	Specifies a file with path maps. This option must be used if your model contains pathmap URIs (URIs that start with “pathmap”, for example `href="pathmap://RT_SAMPLE_LIB/RTSampleCustomLibrary.emx#_FPfjMF83EeiD6r1CftWc8g?RTSampleCustomLibrary/BaseA?"`). See Path Maps for more information.

`--root=<path>`
or
`--root=<map file>`
	Specifies the root folder for the model compiler. All references in model and TC files will be resolved based on this root folder setting. It is therefore usually set to the workspace folder:
	`--root=<path to existing workspace>`
	In this case model compiler will automatically detect location of all projects based on workspace metadata (no matter if they are imported into the workspace from other locations or not).

	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. It is also possible to specify individual locations of all projects in a map file and provide it as an argument for this option. Map file is a text file where location of each project is written on a separate line, like this: `<project-name>=<path-to-project-folder>` Environment variables and system properties can be used inside the map file via $(var) or ${var} syntax. `--root` option can be specified multiple times. All values are merged. The last value overrides previous one.
`--rteSharedLoc=<path>`	This option specifies the location where RSARTE is installed. The model compiler may need this information in case the built model references something that is located in the RSARTE installation.
`--syncMap=<path>`	This option is used when the model compiler is launched by RSARTE 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>`	Sets the validation rules to be performed by the model compiler. The following rules are supported: • `sources` Validate source model elements and report errors if any source element is missing • `tc` TC files are validated. You can set multiple rules separated by a comma. Set the scope to `off` in order to disable all validation.
`--verbose=<level>`	Sets the verbosity of the model compiler. The following verbosity levels are supported: • `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.
`--version`	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.

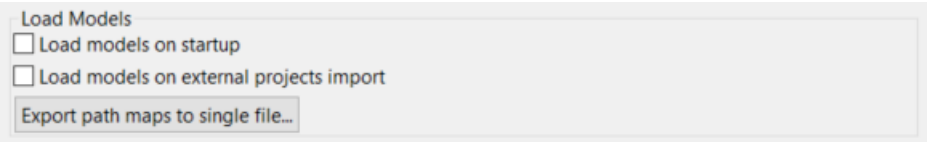
Path Maps

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#_FPfjMF83EeiD6r1CftWc8g?RTSampleCustomLibrary/BaseA?"
```

Path map variables can be defined in the RSARTE 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 RSARTE. There is a button on the *UML 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 RSARTE 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. Command-line option
2. Variable defined in environment file
3. JVM system property
4. 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 RSARTE Eclipse IDE and the model compiler.

The following variables can be used:

Variable	Type	Description
`RTAutoDepsLog`	Boolean	This variable applies only if `RTAutoDeps` 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 *UML Development – Real Time C++ Transformations – Report details about automatically added source elements*.

RTCDRUNVariable	String	Specifies the location of the cdrun.pl script. By default the cdrun.pl from the RSARTE installation is used.
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 *UML Development – Real Time C++ Transformations – Generate type descriptors for complex types*.
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 *UML Development – Real Time C++ Transformations – Context sensitive library builds*.
RTDependVariable	String	Specifies the location of the rtcppdep.pl script. By default the rtcppdep.pl from the RSARTE installation is used.
RTLinkOrder	String	Specifies the default link order for libraries when building an executable. This variable corresponds to the preference *UML Development – Real Time C++ Transformations – Link order*.
RTMakeMode	String	Controls the kind of make file that is generated. It can be either “recursive” (default) or “inclusive”. This variable corresponds to the preference *UML Development – Batch Make Type*. 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.
RTOptimizeMArrays	Boolean	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 *UML Development – Real Time C++ Transformations – Optimize initialization code for multi-dimensional arrays*.
RTPerlVariable	String	Specifies the location of the rtperl executable. By default the rtperl from the RSARTE installation is used.

RTSetupVariable	String	Specifies the location of the rtsetup.pl script. By default the rtsetup.pl from the RSARTE installation is used.
RTToolsVariable	String	Specifies the location of the tools folder. By default the tools folder from the RSARTE installation is used.

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.tc
```

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.tc
```

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.tc
```

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.tc
```

Only generate make files for a TC file.