Build Variants Introduction (Tutorial)

Author: Elena Volkova
Technical Specialist
IBM

In this tutorial we will demo how to design a build system for a model using build variants:

- define different variants for building a transformation configuration
- define user interface for the build command
- use build variants when building for different targets

System Requirements:

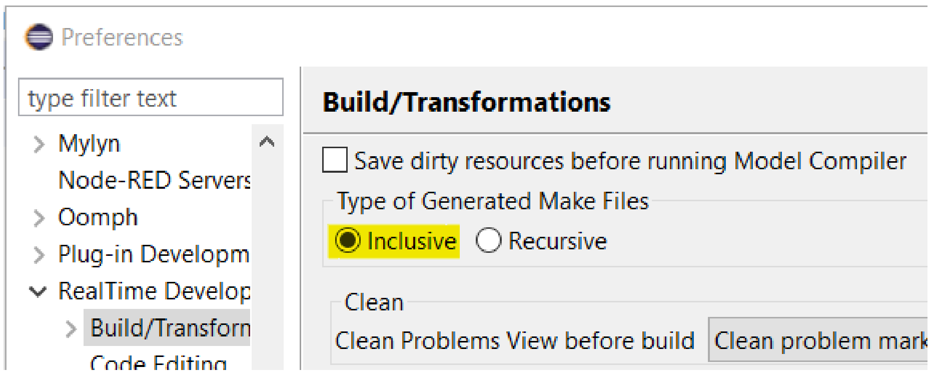
- Windows 10 + MinGW
- DevOps Model RealTime 10.2 2018.21 or later

This document was last updated for DevOps Model RealTime 11.0.

Preparation

- Create a new folder `WS_BuildVariants_Tutorial` for a new Model RealTime workspace
- Open Model RealTime with the new workspace
  - eclipse.exe -data C:/_Work/tmp_test/WS_BuildVariants_Tutorial
- Import projects from `BuildVariants_Tutorial.zip` to the new workspace
- Set the following in the workspace preferences:

  - RealTime Development → Build/Transformations → Type of Generated Makefile = Inclusive



Build/Transformations

☐ Save dirty resources before running Model Compiler

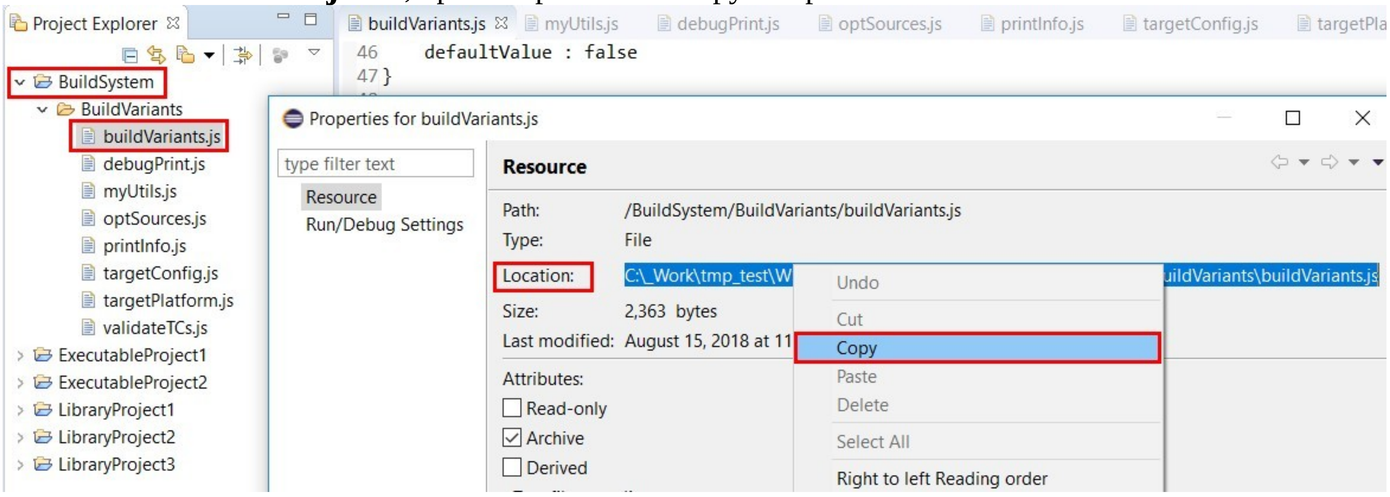
Type of Generated Make Files

[✓ Inclusive] Recursive

Clean

Clean Problems View before build: Clean problem mark

- In the project `BuildSystem` under the folder `BuildVariants` right-click on **buildVariants.js** file, open Properties and copy full path to the file



Project Explorer

BuildSystem
BuildVariants
buildVariants.js
debugPrint.js
myUtils.js
optSources.js
printInfo.js
targetConfig.js
targetPlatform.js
validateTCs.js
ExecutableProject1
ExecutableProject2
LibraryProject1
LibraryProject2
LibraryProject3

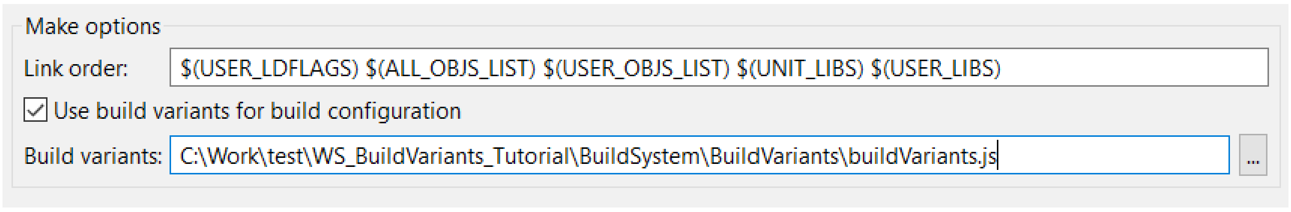
```
46      defaultValue : false
47 }
```

Properties for buildVariants.js

Resource

Path: /BuildSystem/BuildVariants/buildVariants.js
Type: File
Location: C:_Work\tmp_test\W…uildVariants\buildVariants.js
Size: 2,363 bytes
Last modified: August 15, 2018 at 11
Attributes:
☐ Read-only
☑ Archive
☐ Derived

- Turn ON build variants for build configuration in the Model RealTime preferences:
  - RealTime Development → Build/Transformations → C++
    - Use build variants for build configuration
    - Build variants: Full path to **buildVariants.js** (copied in the previous step)



Make options

Link order: $(USER_LDFLAGS) $(ALL_OBJS_LIST) $(USER_OBJS_LIST) $(UNIT_LIBS) $(USER_LIBS)

☑ Use build variants for build configuration

Build variants: C:\Work\test\WS_BuildVariants_Tutorial\BuildSystem\BuildVariants\buildVariants.js

Contents

In the project `BuildSystem` under the folder `BuildVariants` you will find the following files:

- **buildVariants.js** - this is the file with build variants declaration. It defines user interface and build variants that will be visible to end users. For each build variant it specifies its implementation script that will modify general transformation settings on the fly to get build settings for a concrete build variant (target platform)
- **myUtils.js** - this is the file with user-defined helpful utils, like appendValue or printSomeProperties
- All other js files are implementation of build variants. Most of them are using functions from myUtils.js
  - **targetPlatform.js**
  - **targetConfig.js**
  - **debugPrint.js**
  - **optSources.js**
  - **printInfo.js**
  - **validateTCs.js**

Declaring build variants

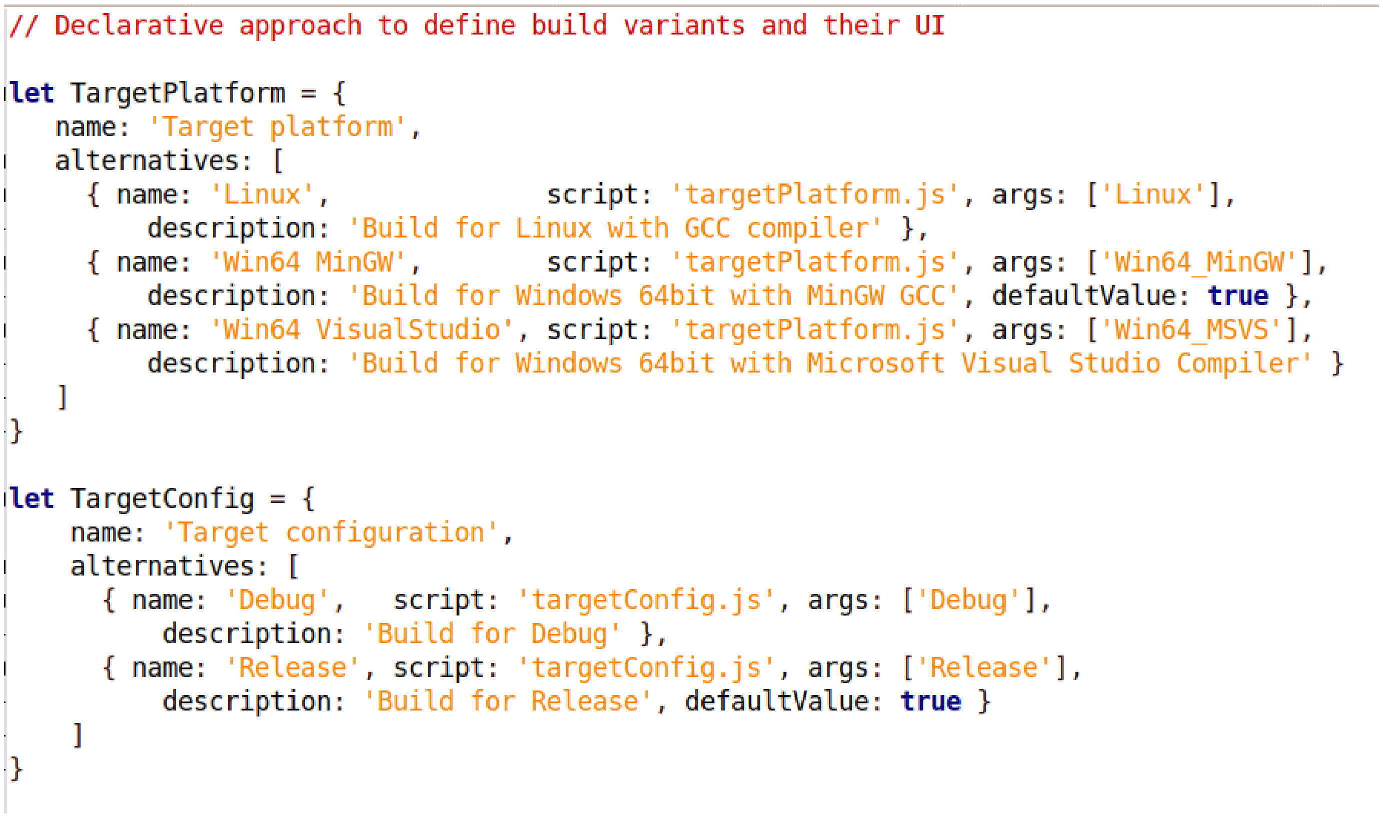
A build engineer should identify different build configurations that users can apply to their models. In **buildVariants.js** file he declares all different kind of build variants by defining:

- name
- script – path to js file with build variant implementation, where settings are actually applied to the TC
- description – textual description of build variant for end users
- alternatives – if there are several alternative settings
- defaultValue – if build variants should be selected by default

For this tutorial we have identified the following build variants for our build system:

- Target platform – Select target platform and compiler. One of these:
  - Linux
  - Win64 MinGW
  - Win64 VisualStudio
- Target configuration – Debug/Release configuration for build. One of these:
  - Debug
  - Release
- Debug print – Build generated code with additional debug printing
- Sources optimization – ON / OFF -- turn ON or OFF sources optimization
- Print TC info – Print information about some TC settings and print all prerequisites
- GNU Coverage – Compile generated code for GNU Coverage analysis
- Validate TCs – Validate user settings in TCs to follow team guidelines and naming conventions

Check the declarations of build variants in **buildVariants.js**:



```
// Declarative approach to define build variants and their UI

let TargetPlatform = {
   name: 'Target platform',
   alternatives: [
     { name: 'Linux',             script: 'targetPlatform.js', args: ['Linux'],
         description: 'Build for Linux with GCC compiler' },
     { name: 'Win64 MinGW',       script: 'targetPlatform.js', args: ['Win64_MinGW'],
         description: 'Build for Windows 64bit with MinGW GCC', defaultValue: true },
     { name: 'Win64 VisualStudio', script: 'targetPlatform.js', args: ['Win64_MSVS'],
         description: 'Build for Windows 64bit with Microsoft Visual Studio Compiler' }
   ]
}

let TargetConfig = {
    name: 'Target configuration',
    alternatives: [
      { name: 'Debug',   script: 'targetConfig.js', args: ['Debug'],
          description: 'Build for Debug' },
      { name: 'Release', script: 'targetConfig.js', args: ['Release'],
          description: 'Build for Release', defaultValue: true }
    ]
}
```

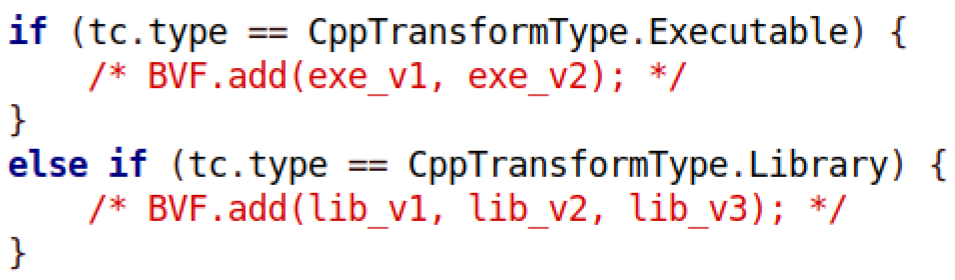
Build variants, that will be visible for a concrete TC are initialized by `initBuildVariants(tc)` function that is required to be implemented in build variants declarations file **buildVariants.js:**

```
function initBuildVariants(tc) {
  BVF.add(TargetPlatform);
  BVF.add(TargetConfig);
  BVF.add(DebugPrint);
  BVF.add(sourceOptimization);
  BVF.add(ValidateTCs);
  BVF.add(Info);

  if (tc.type == CppTransformType.Executable) {
  } else {
  }

}
```

`BVF.add(declaredVariant);` is used to add this build variant to the list of available build variants for a concrete TC.

So it is possible to specify different set of build variants for different TCs. One of the most common usages is to distinguish the set of available build variants between Library TCs and Executable TCs.

You can check the kind of a TC and add specific build variants in the if-else sections :



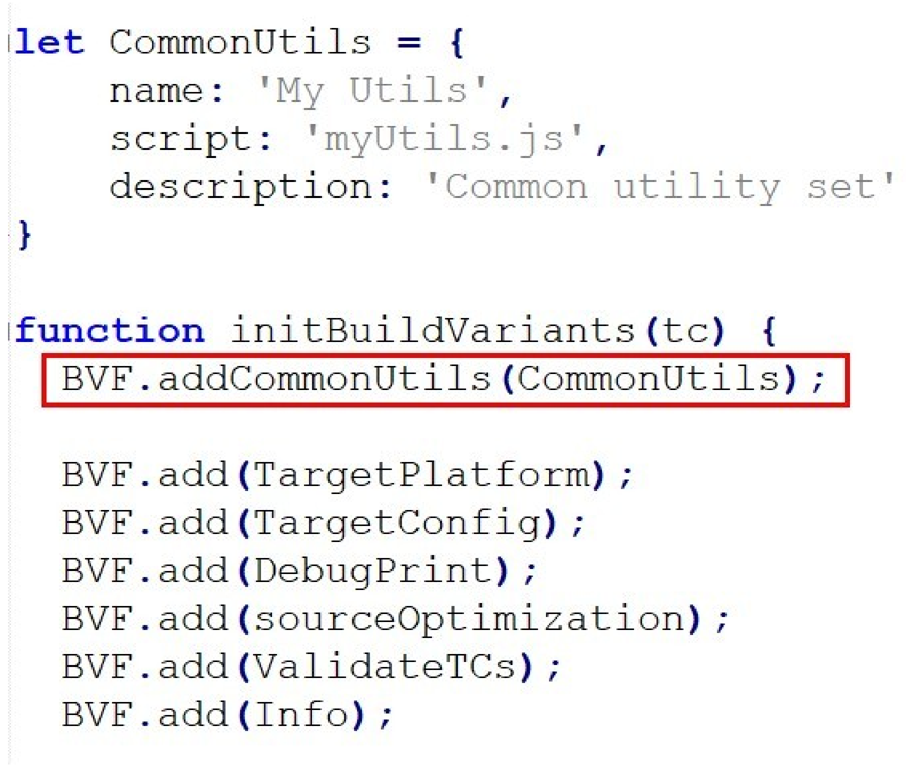
```
if (tc.type == CppTransformType.Executable) {
    /* BVF.add(exe_v1, exe_v2); */
}
else if (tc.type == CppTransformType.Library) {
    /* BVF.add(lib_v1, lib_v2, lib_v3); */
}
```

User-defined utility functions for build variants implementation

It is always useful to define some user-specific helpful functions to manipulate TCs that can be referenced from JavaScript build variant implementation files.
The file **myUtils.js** is used for this purpose. It contains functions such as:

- appendValue – append a new argument to the end of a string TC property
- CWD – print current working directory
- printSomeProperties(tc) – print the values of several most-interesting TC properties

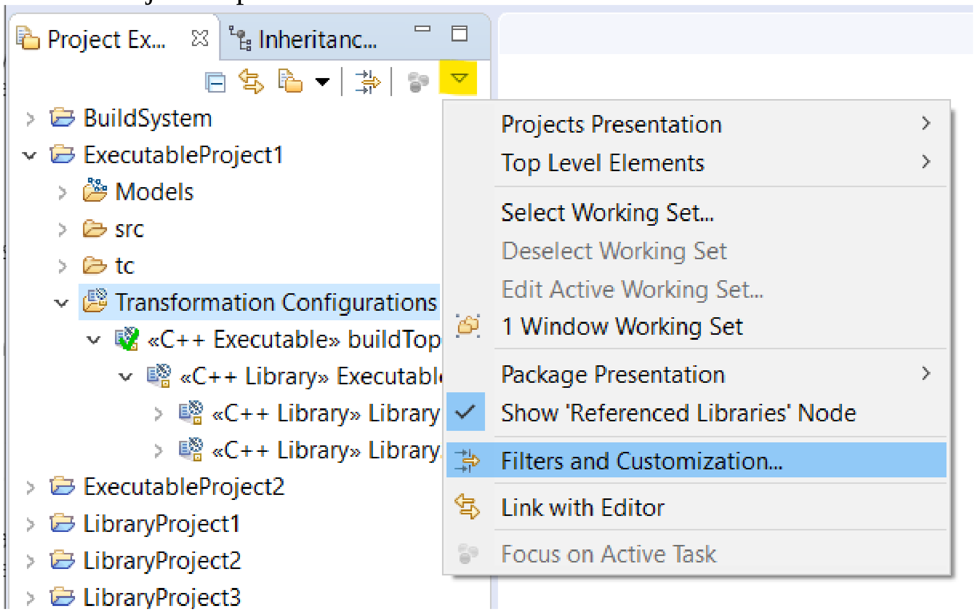
To make the functions from **myUtils.js** available in all build variants implementation files, an object representing the utility set should be declared and then added to the build variants with `BVF.addCommonUtils` function inside `initBuildVariants(tc)`:



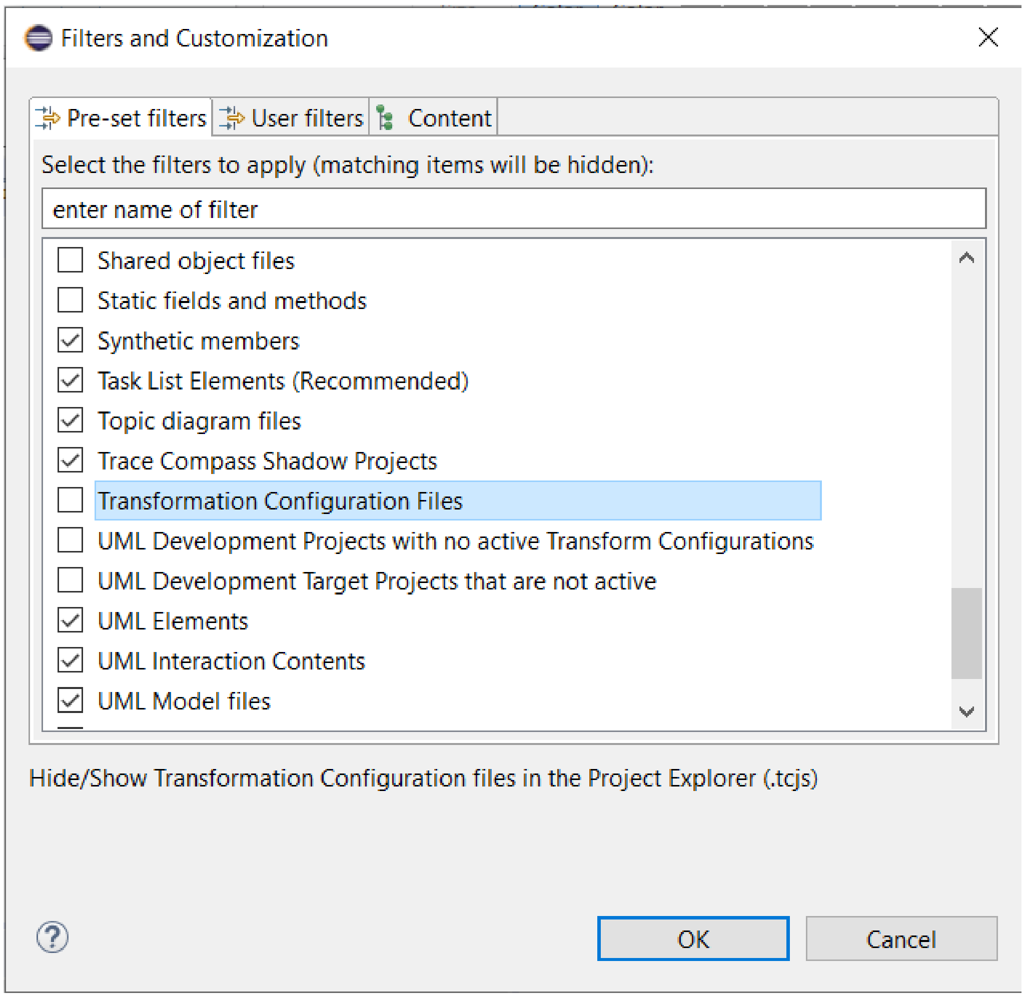
```
let CommonUtils = {
    name: 'My Utils',
    script: 'myUtils.js',
    description: 'Common utility set'
}

function initBuildVariants(tc) {
  BVF.addCommonUtils(CommonUtils);

  BVF.add(TargetPlatform);
  BVF.add(TargetConfig);
  BVF.add(DebugPrint);
  BVF.add(sourceOptimization);
  BVF.add(ValidateTCs);
  BVF.add(Info);
```

User interface for build variants

All transformation configuration files are defined in the *.tcjs textual (JavaScript) format. Corresponding TC nodes are visible under the `Transformation Configurations` virtual folder in the Project Explorer:



By default TC files are hidden in the Project Explorer, you see only TC nodes under the virtual folder. To see file names, update `Filters and Customization`... settings and deselect `Transformation Configuration Files`:


Filters and Customization

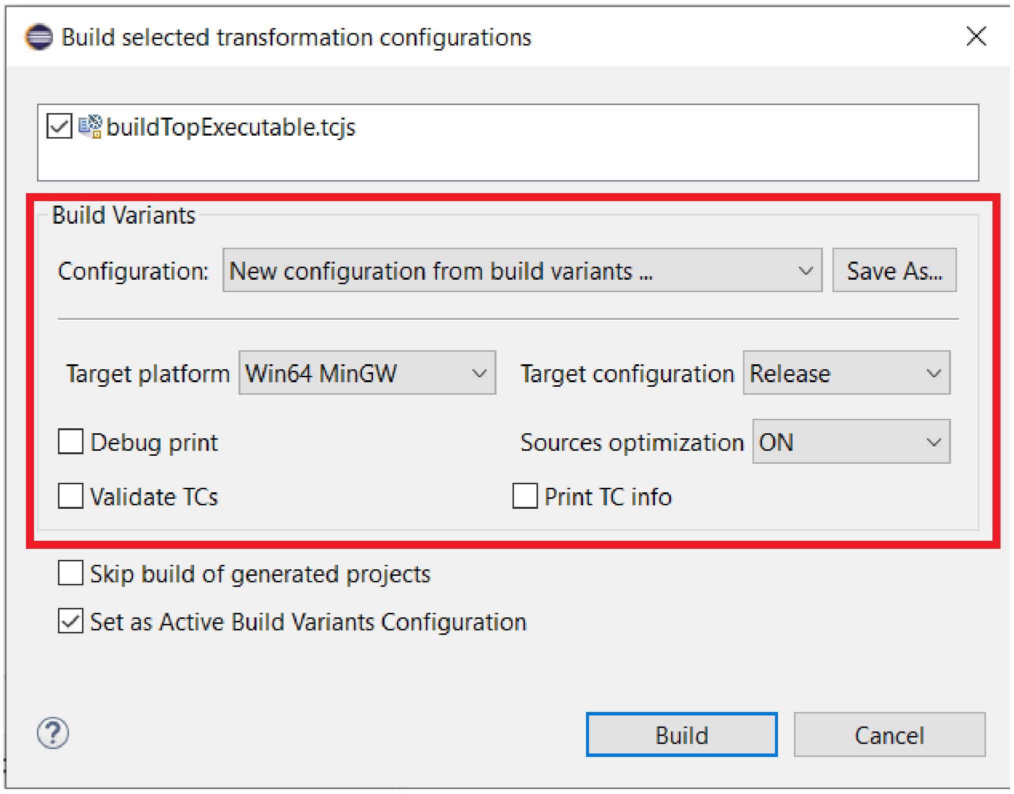
Select the filters to apply (matching items will be hidden):

enter name of filter

- [ ] Shared object files
- [ ] Static fields and methods
- [x] Synthetic members
- [x] Task List Elements (Recommended)
- [x] Topic diagram files
- [x] Trace Compass Shadow Projects
- [ ] Transformation Configuration Files
- [ ] UML Development Projects with no active Transform Configurations
- [ ] UML Development Target Projects that are not active
- [x] UML Elements
- [x] UML Interaction Contents
- [x] UML Model files

Hide/Show Transformation Configuration files in the Project Explorer (.tcjs)

Right-click on the TC `buildTopExecutable` (either on the TC file or the TC node in the virtual folder) and select `Build...` – all available build variants will appear in the user interface:



Build selected transformation configurations

- [✓] buildTopExecutable.tcjs

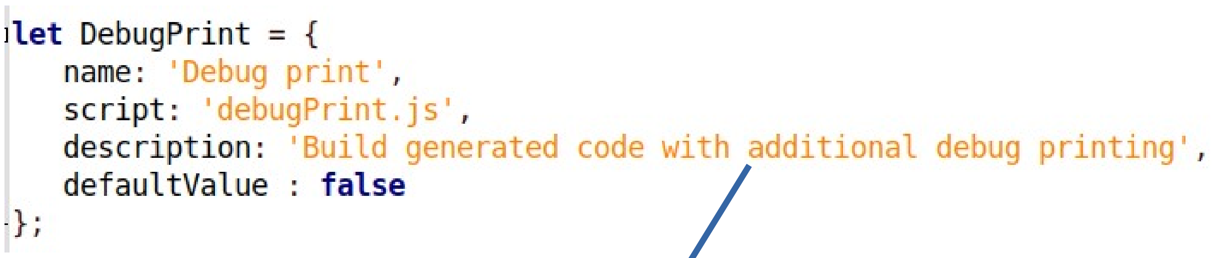
Build Variants

Configuration: New configuration from build variants ... | Save As...

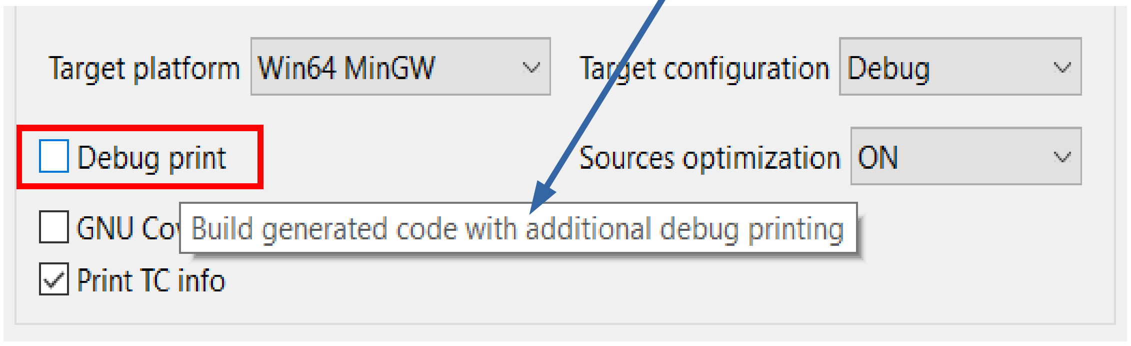
Target platform: Win64 MinGW
Target configuration: Release
- [ ] Debug print
- Sources optimization: ON
- [ ] Validate TCs
- [ ] Print TC info

- [ ] Skip build of generated projects
- [✓] Set as Active Build Variants Configuration

The user will see the descriptions specified in the **buildVariants.js** for each build variant when holding a mouse pointer over the UI section:



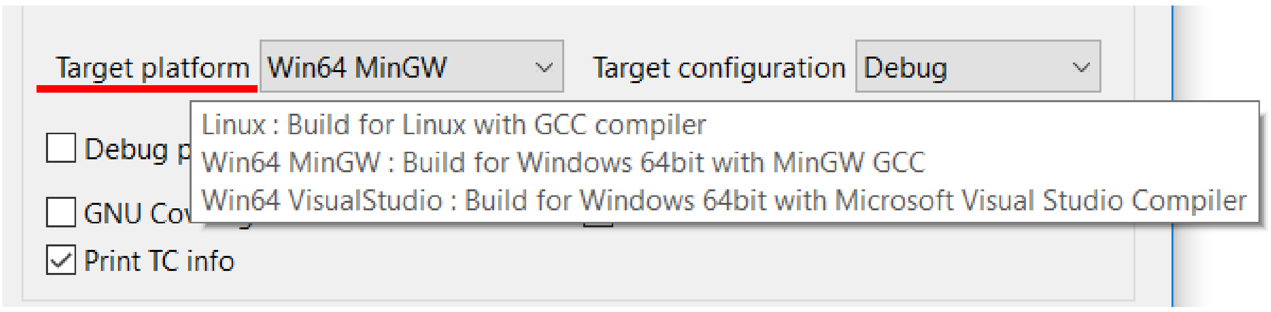
```
let DebugPrint = {
    name: 'Debug print',
    script: 'debugPrint.js',
    description: 'Build generated code with additional debug printing',
    defaultValue : false
};
```



Target platform: Win64 MinGW
Target configuration: Debug
- [ ] Debug print
- Sources optimization: ON
- [ ] GNU Co
- [✓] Print TC info

Build generated code with additional debug printing

For build variants with alternatives you can see descriptions of all alternatives:



Target platform: Win64 MinGW
Target configuration: Debug

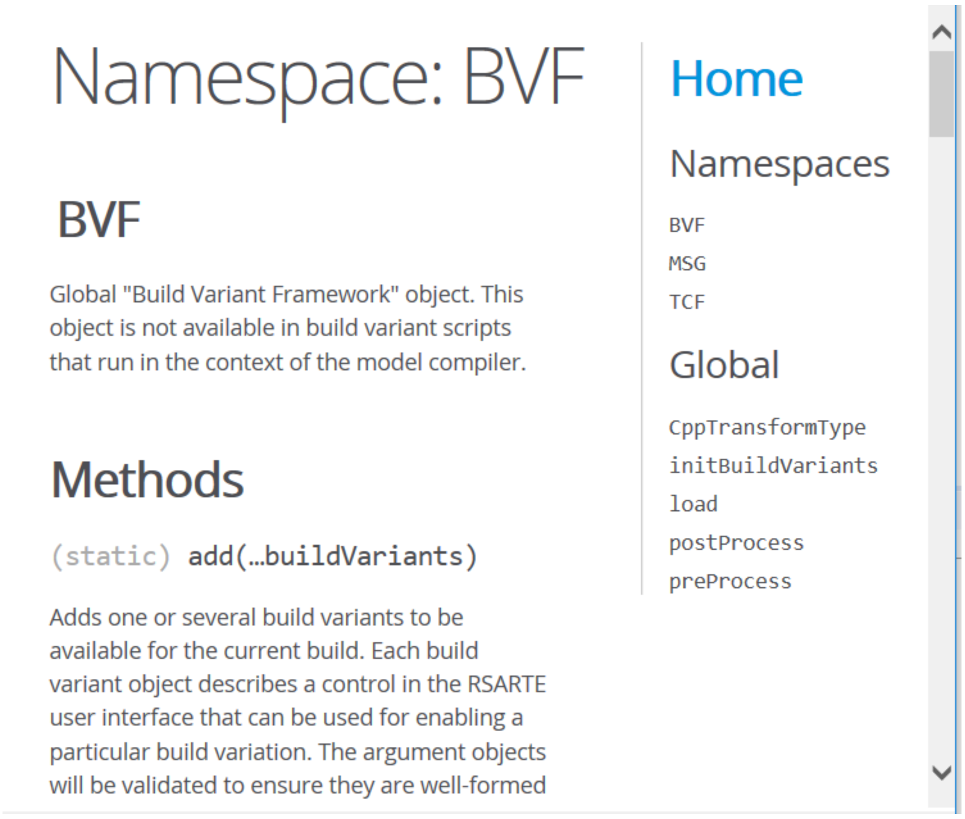
- Linux : Build for Linux with GCC compiler
- Win64 MinGW : Build for Windows 64bit with MinGW GCC
- Win64 VisualStudio : Build for Windows 64bit with Microsoft Visual Studio Compiler

☐ Debug p
☐ GNU Co
☑ Print TC info

Build variants API Reference

API functions and data types that can be used inside build variants declaration file and implementation files are described in the tool documentation.
Find API Reference under Help → Help Contents → Model RealTime Java APIs → Model RealTime Transformation Developer's Guide. Click on the link called "Transformation Configuration and Build Variants JavaScript API".

Basic functions to use build variants are documented:



Namespace: BVF

BVF

Global "Build Variant Framework" object. This object is not available in build variant scripts that run in the context of the model compiler.

Methods

(static) add(…buildVariants)

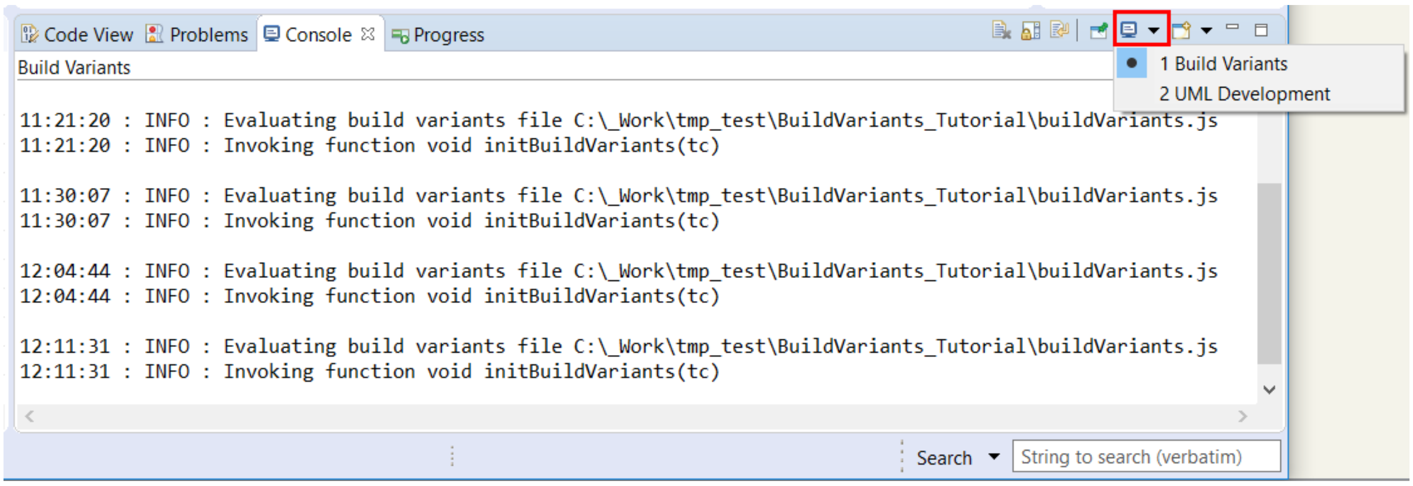
Adds one or several build variants to be available for the current build. Each build variant object describes a control in the RSARTE user interface that can be used for enabling a particular build variation. The argument objects will be validated to ensure they are well-formed

Build Variants Console and message printing

The build variants declarations file is evaluated each time a TC is built. All the updates in that file are immediately reflected in the UI. But the end-user is not supposed to do any updates to the build variants. This is a task for the experienced user – the build engineer, who is designing the build system to be used by end users.

All the messages from build variants evaluation are printed in a Build Variants Console, but it will not pop-up for the user unless there are some problems with interpreting the build variants declaration script (**buildVariants.js**).

Check the build variants console and default messages in it:



Build Variants

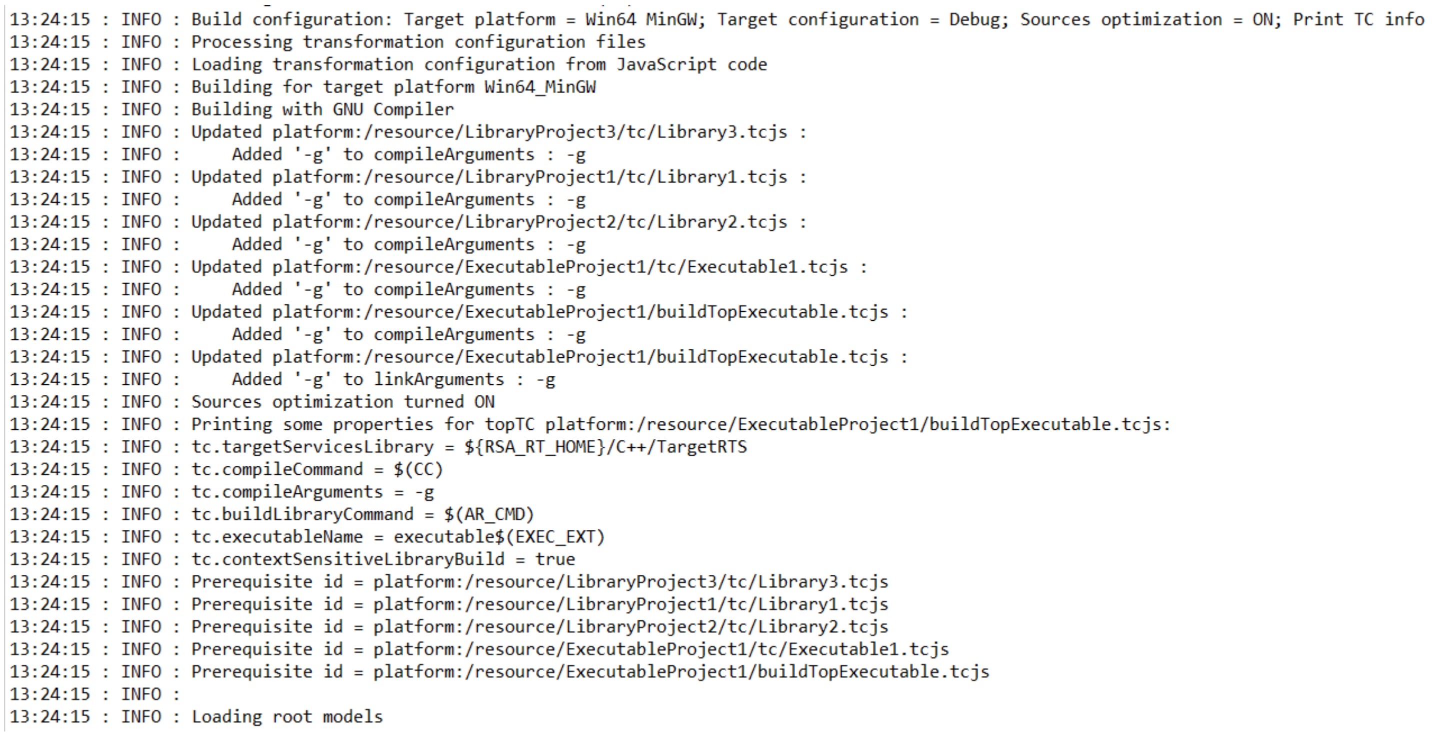
```
11:21:20 : INFO : Evaluating build variants file C:\_Work\tmp_test\BuildVariants_Tutorial\buildVariants.js
11:21:20 : INFO : Invoking function void initBuildVariants(tc)

11:30:07 : INFO : Evaluating build variants file C:\_Work\tmp_test\BuildVariants_Tutorial\buildVariants.js
11:30:07 : INFO : Invoking function void initBuildVariants(tc)

12:04:44 : INFO : Evaluating build variants file C:\_Work\tmp_test\BuildVariants_Tutorial\buildVariants.js
12:04:44 : INFO : Invoking function void initBuildVariants(tc)

12:11:31 : INFO : Evaluating build variants file C:\_Work\tmp_test\BuildVariants_Tutorial\buildVariants.js
12:11:31 : INFO : Invoking function void initBuildVariants(tc)
```

Build variants implementation files can also print messages when they are evaluated by references functions from MSG global namespace. For example, check targetPlatform.js, targetConfig.js, myUtils.js, **printInfo.js** :

```
function postProcess(topTC, allTCs) {
    MSG.formatInfo("Printing some properties for topTC %s:", topTC.getId());
    printSomeProperties(topTC.eval);
    for (i = 0; i < allTCs.length; ++i) {
        MSG.formatInfo("Prerequisite id = %s", allTCs[i].getId());
    }
    MSG.formatInfo("");
}
```

Also check API Reference for MSG namespace and its available functions.

All these messages will appear in the build log:



```
13:24:15 : INFO : Build configuration: Target platform = Win64 MinGW; Target configuration = Debug; Sources optimization = ON; Print TC info
13:24:15 : INFO : Processing transformation configuration files
13:24:15 : INFO : Loading transformation configuration from JavaScript code
13:24:15 : INFO : Building for target platform Win64_MinGW
13:24:15 : INFO : Building with GNU Compiler
13:24:15 : INFO : Updated platform:/resource/LibraryProject3/tc/Library3.tcjs :
13:24:15 : INFO :     Added '-g' to compileArguments : -g
13:24:15 : INFO : Updated platform:/resource/LibraryProject1/tc/Library1.tcjs :
13:24:15 : INFO :     Added '-g' to compileArguments : -g
13:24:15 : INFO : Updated platform:/resource/LibraryProject2/tc/Library2.tcjs :
13:24:15 : INFO :     Added '-g' to compileArguments : -g
13:24:15 : INFO : Updated platform:/resource/ExecutableProject1/tc/Executable1.tcjs :
13:24:15 : INFO :     Added '-g' to compileArguments : -g
13:24:15 : INFO : Updated platform:/resource/ExecutableProject1/buildTopExecutable.tcjs :
13:24:15 : INFO :     Added '-g' to compileArguments : -g
13:24:15 : INFO : Updated platform:/resource/ExecutableProject1/buildTopExecutable.tcjs :
13:24:15 : INFO :     Added '-g' to linkArguments : -g
13:24:15 : INFO : Sources optimization turned ON
13:24:15 : INFO : Printing some properties for topTC platform:/resource/ExecutableProject1/buildTopExecutable.tcjs:
13:24:15 : INFO : tc.targetServicesLibrary = ${RSA_RT_HOME}/C++/TargetRTS
13:24:15 : INFO : tc.compileCommand = $(CC)
13:24:15 : INFO : tc.compileArguments = -g
13:24:15 : INFO : tc.buildLibraryCommand = $(AR_CMD)
13:24:15 : INFO : tc.executableName = executable$(EXEC_EXT)
13:24:15 : INFO : tc.contextSensitiveLibraryBuild = true
13:24:15 : INFO : Prerequisite id = platform:/resource/LibraryProject3/tc/Library3.tcjs
13:24:15 : INFO : Prerequisite id = platform:/resource/LibraryProject1/tc/Library1.tcjs
13:24:15 : INFO : Prerequisite id = platform:/resource/LibraryProject2/tc/Library2.tcjs
13:24:15 : INFO : Prerequisite id = platform:/resource/ExecutableProject1/tc/Executable1.tcjs
13:24:15 : INFO : Prerequisite id = platform:/resource/ExecutableProject1/buildTopExecutable.tcjs
13:24:15 : INFO :
13:24:15 : INFO : Loading root models
```

Specifying target platform and target configuration during build – targetPlatform.js and targetConfig.js

These build variants will set correct make type and target configuration for top TC and all prerequisites. For Debug will also apply Debug compiler flags.

Steps:

- Make clean
- Build with "Win64 MinGW” + “Release" build variants
- check executable size WS_BuildVariants_Tutorial\ExeWithEmptySources_target\default\executable.EXE - it is small as it does not contain debug information
- Make clean
- Build with "Win64 MinGW” + “Debug" => debug flags will be applied on the fly
- Check size of WS_BuildVariants_Tutorial\ExeWithEmptySources_target\default\executable.EXE - now it is bigger because we added debug information. It can be used for debug.

Try to extend these build variants for more platforms and compilers.

Sources optimization with build variant – optSources.js

This build variant will turn ON or OFF filtering of sources (context sensitive library builds)

Steps:

- Make clean
- Try with ON - build time will be faster and in the log there are less sources for build
- Make clean
- Try with OFF - build will run slower and all elements from UML model will be generated and compiled, no matter if they are referenced from top capsule or not

Printing specific TC info during build – printInfo.js

This build variant will print information to the log about actual values (including default values and inheritance) for some TC properties, for example:

```
22:54:59 : INFO : Printing some properties for topTC
platform:/resource/ExecutableProject1/buildTopExecutable.tcjs:
22:54:59 : INFO : tc.targetServicesLibrary =
${RSA_RT_HOME}/C++/TargetRTS
22:54:59 : INFO : tc.compileCommand = $(CC)
22:54:59 : INFO : tc.compileArguments = -g
22:54:59 : INFO : tc.buildLibraryCommand = $(AR_CMD)
22:54:59 : INFO : tc.executableName = executable$(EXEC_EXT)
22:54:59 : INFO : Prerequisite id =
platform:/resource/LibraryProject3/tc/Library3.tcjs
22:54:59 : INFO : Prerequisite id =
platform:/resource/LibraryProject1/tc/Library1.tcjs
22:54:59 : INFO : Prerequisite id =
platform:/resource/LibraryProject2/tc/Library2.tcjs
22:54:59 : INFO : Prerequisite id =
platform:/resource/ExecutableProject1/tc/Executable1.tcjs
22:54:59 : INFO : Prerequisite id =
platform:/resource/ExecutableProject1/buildTopExecutable.tcjs
```

Enable custom Debug printing – debugPrint.js

This build variant will add a preprocessor define DEBUG_PRINT to compile flags and so enable additional print statements defined in the model

Steps:

- Make clean
- Build executable with "Debug print" build variant
- Check that generated batch.mk contains -DDEBUG_PRINT for compilation of all TCs
- Double-click on WS_BuildVariants_Tutorial\ExeWithEmptySources_target\default\executable.EXE
  - RTS debug: ->
  - type "quit", enter
- You will see the following debug print on the screen:

Initializing CapsuleLibrary1_blue statemachine

Initializing CapsuleLibrary2_blue statemachine
Initializing Capsule1_blue statemachine

- Press Ctrl-C to stop execution

These statements are surrounded by conditional compilation macros #ifdef DEBUG_PRINT ... #endif and will be executed only when "Debug print" build variant is enabled.

- Make clean
- Build executable without "Debug print".
- Double-click on WS_BuildVariants_Tutorial\ExeWithEmptySources_target\default\executable.EXE
    - RTS debug: →
    - type "quit", enter
- Nothing is printed on the screen.
- Press Ctrl-C to stop execution

Validate user settings in TCs – validateTCs.js

This build variant will check that user is following specific naming conventions and print error messages in case of problems.

Steps:

- Build executable and select “Validate TCs” build variant
- This option will add a check for the name of target project, it should start with prefix “target_”
- Build will finish with the error messages complaining that for some TCJS files target project name does not start with “target_”:

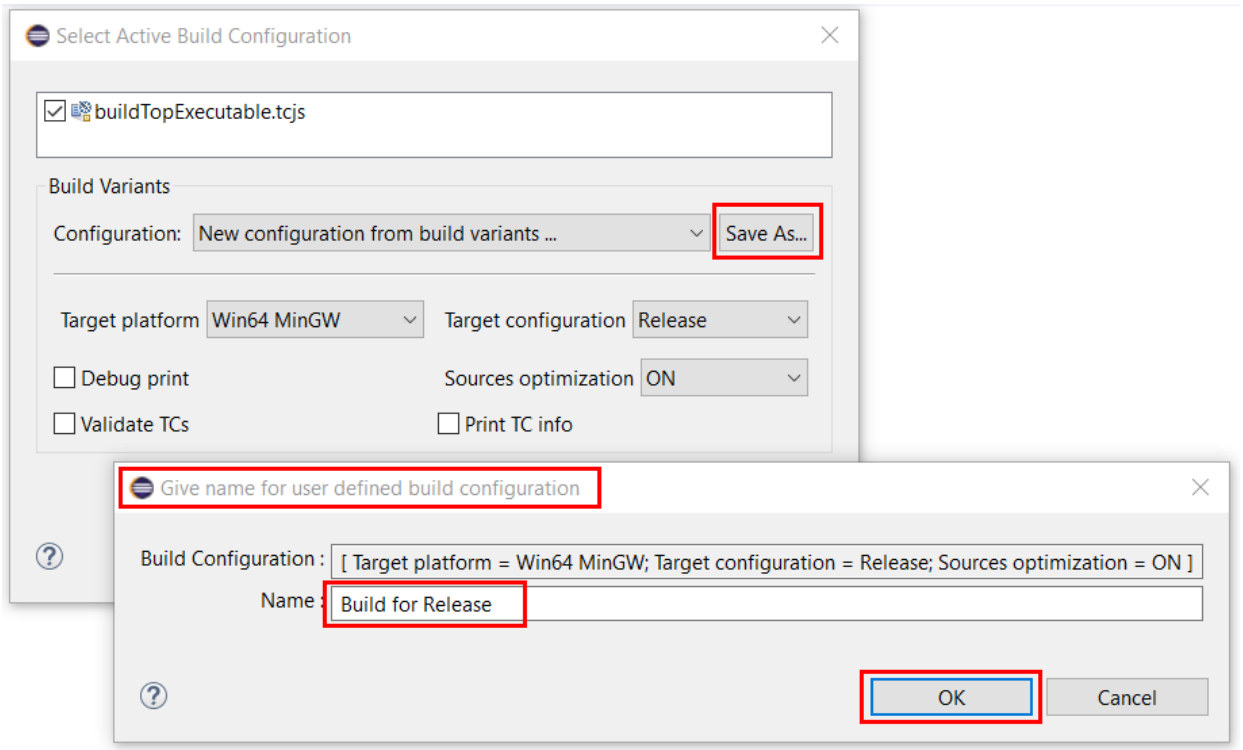
```
16:23:11 : INFO : Sources optimization turned OFF
16:23:11 : ERROR : Target project for
platform:/resource/LibraryProject3/tc/Library3.tcjs does not
start with 'target_' : /Library3_native
16:23:11 : ERROR : Target project for
platform:/resource/LibraryProject1/tc/Library1.tcjs does not
start with 'target_' : /Library1_native
16:23:11 : ERROR : Target project for
platform:/resource/LibraryProject2/tc/Library2.tcjs does not
start with 'target_' : /Library2_native
16:23:11 : ERROR : Target project for
platform:/resource/ExecutableProject1/tc/Executable1.tcjs
does not start with 'target_' : /Executable1_native
16:23:11 : INFO : Target project for
platform:/resource/ExecutableProject1/buildTopExecutable.tcjs
starts with 'target_' : target_TopExecutable
16:23:11 : ERROR : Skipping loading root models because of
errors
16:23:11 : INFO : Done. Elapsed time 0.16 s
```

Creating user-defined build configurations

For some basic sets of build variants it is useful to save them under some name.

Steps:

- Right click on `buildTopExecutable` and select `Configure Build Variants…`
- A dialog with build variants will pop-up
- Select the following build variants and click on `Save As…`



Select Active Build Configuration

☑ buildTopExecutable.tcjs

Build Variants

Configuration: New configuration from build variants ... | Save As...

Target platform: Win64 MinGW | Target configuration: Release

☐ Debug print | Sources optimization: ON

☐ Validate TCs | ☐ Print TC info

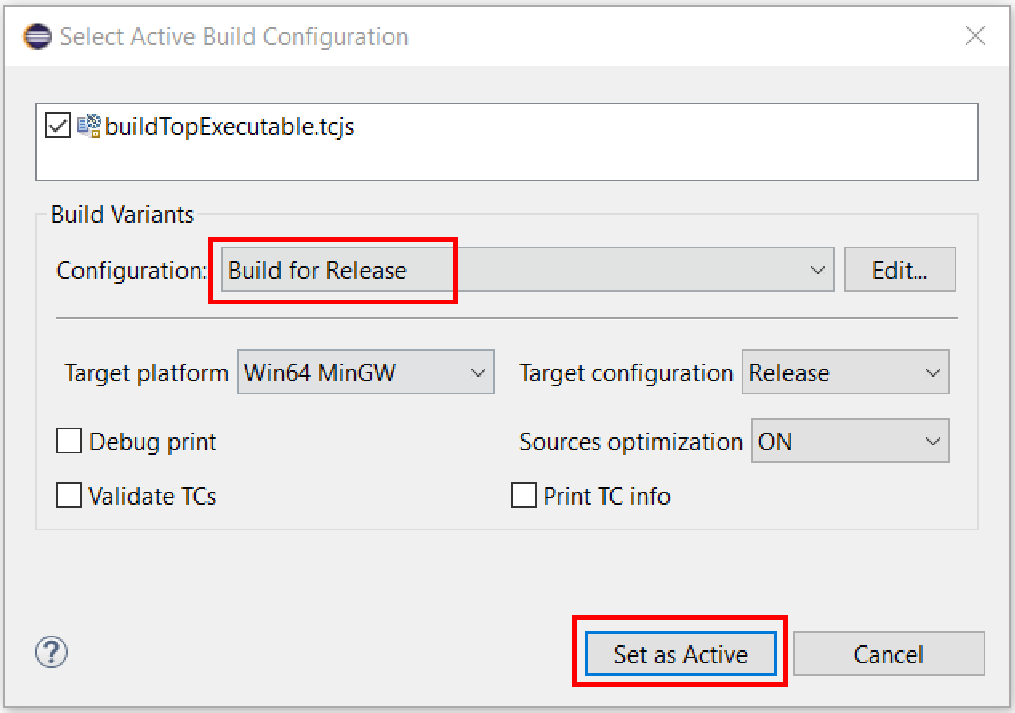
Give name for user defined build configuration

Build Configuration : [Target platform = Win64 MinGW; Target configuration = Release; Sources optimization = ON]

Name : Build for Release

OK | Cancel

- Type a new name for the selected set of build variants: `Build for Release`
- Press `OK`
- You will see that given name is saved under `Configuration` drop-down list



Select Active Build Configuration

☑ buildTopExecutable.tcjs

Build Variants

Configuration: Build for Release | Edit...

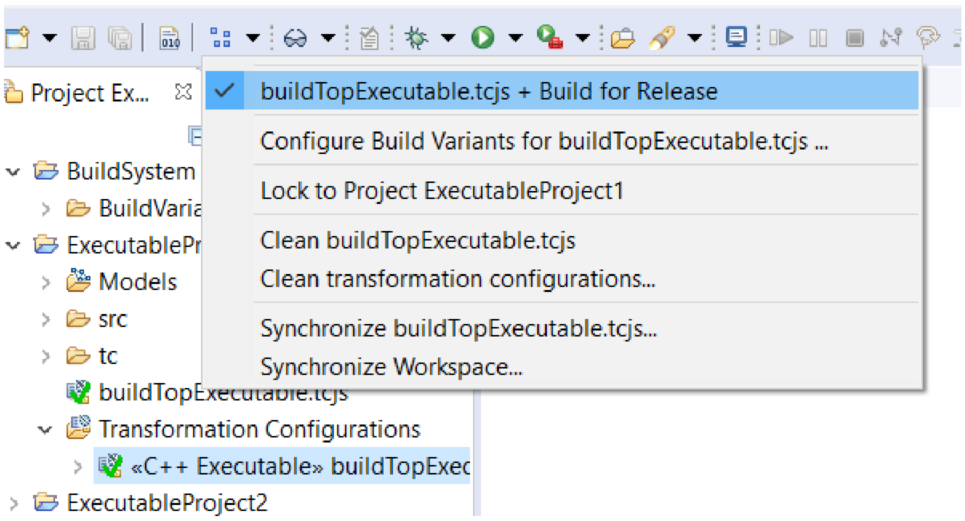
Target platform: Win64 MinGW | Target configuration: Release

☐ Debug print | Sources optimization: ON

☐ Validate TCs | ☐ Print TC info

Set as Active | Cancel

- Press `Set as Active` to make this configuration active by default for the `buildTopExecutable` TC
- Open the `Build Active Transformation Configuration` drop-down menu and see that this configuration is active for `buildTopExecutable` and will be applied by default when you press the `Build Active Transformation Configuration` toolbar button



Project Ex...
BuildSystem
BuildVaria
ExecutablePr
Models
src
tc
buildTopExecutable.tcjs
Transformation Configurations
«C++ Executable» buildTopExec
ExecutableProject2
✓ buildTopExecutable.tcjs + Build for Release
Configure Build Variants for buildTopExecutable.tcjs ...
Lock to Project ExecutableProject1
Clean buildTopExecutable.tcjs
Clean transformation configurations...
Synchronize buildTopExecutable.tcjs...
Synchronize Workspace...

- Press the `Build Active Transformation Configuration` toolbar button. The `Build for Release` configuration will be applied to `buildTopExecutable`.

```
16:09:42 : INFO : Build configuration: Target platform = Win64
MinGW; Target configuration = Release; Sources optimization = ON
16:09:42 : INFO : Processing transformation configuration files
16:09:42 : INFO : Loading transformation configuration from
JavaScript code
16:09:42 : INFO : Building for target platform Win64_MinGW
16:09:42 : INFO : Building with GNU Compiler
16:09:42 : INFO : Sources optimization turned ON
16:09:42 : INFO : Loading root models
```

- Create more configurations from Build Variants and give them new names
- For example
  - `Build for Debug` = [Target platform = Win64 MinGW; Target configuration = Debug; Debug print; Sources optimization = ON; Print TC info]
  - `Validate according to Team naming conventions` = [Target platform = Win64 MinGW; Target configuration = Release; Sources optimization = OFF; Validate TCs; Print TC info]

- Next time you select `Build`... you will see all these names in the drop-down list of Configurations and you can select any of them as Active for your TC