Managing Build Configuration Complexity in Industrial Embedded Systems

- Dynamic Manipulation of Model Transformations using JavaScript

Mattias Mohlin and Elena Strabykina, HCL Technologies
CONTENT

- UML-RT models and transformation configurations
- Build Variants
- Demo
TRANSFORMING UML-RT MODELS TO C++

- A UML-RT model with contained C++ code is a complete specification of an application (e.g. embedded or IoT application)

- A Transformation Configuration is a model describing how to transform the UML-RT model to a C++ executable or library
UML-RT MODEL

- Behavior described by state machine diagrams and C++ code
- Structure described by class and composite structure diagrams
- Stored in XML files with embedded C++ code snippets
TRANSFORMATION CONFIGURATION

- A model containing everything needed for transforming the UML-RT model into a C++ program and building an executable or library from it.
- Stored in text files using the JavaScript language.
BUILDING MULTIPLE VARIANTS OF AN APPLICATION

- **Problem:** How to express variability in a transformation configuration?
  - Debug vs Release version
  - Different target platforms (OS, compiler etc.)
  - Instrumented builds (e.g. purify)
  - Static analysis (e.g. lint), etc...

- With static transformation configurations, you need one for each variant you want to build...
  => A huge number of transformation configurations to create and maintain!
  => Difficult for users to pick a consistent set of transformation configurations when building the model!

- Inheritance allows to break out common information in separate transformation configurations, but doesn’t solve the problem (still a huge number of transformation configurations even if they all are small).

- We need dynamic transformation configurations where build properties can be manipulated programmatically!
BUILD VARIANTS

- **Solution:** Allow the transformation configuration to be dynamically manipulated at build time

**Build Variant** - a set of transformation configuration properties that are specific for a certain type of build, described in a separate JavaScript file.

```javascript
let globals = TCF.globals();
globals.makeArguments = '$ARG_GLOBAL';

let tc = TCF.define(TCF.CPP_TRANSFORM);
tc.sources = ['platform:/resource/CM/CPPModel.emf#_uysz8NQ3EexPbULy_rI8g'];
tc.genUserCodeQualifiers = globals.makeArguments + 'MY_ARGUMENT';
tc.type = CppTransformType.Executable;

tc.makeType = 'Library_makeType';
tc.TargetRTS = '{RSA_RT_HOME}/C++/TargetRTS';

tc.compileArguments = 'new_compile_arguments';

let ext_TC = TCF.load('platform:/resource/Lib/lib.tc');
tc.prerequisites = ext_TC.prer.add('platform:/resource/LibraryProject1/tc/Lib2.js');
```
IMPLEMENTATION OF BUILD VARIANTS APPROACH

- Each build variant script can be called one or two times, by defining one or both of these functions
  - function preProcess(<args>)
    Called before evaluation of the transformation configuration. Set global properties that can be referenced later.
  - function postProcess(topTC, allTCs, <args>)
    Called after evaluation of the transformation configuration. Verify and override user-defined properties.

- Custom arguments can be passed from the build variants script (allows to reuse the same build variant script for multiple choices)
INTEGRATION WITH IDE

- A build variants declaration file (JavaScript) describes the high-level choices that cause the variability.

- The script renders a dynamic user interface to allow the user to make the choices when building.
  - Checkbox for single choice
  - Drop-down menu for multiple choice

- Each choice is mapped to a build variant script to be applied to the transformation configuration when it is built.

```javascript

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)
}
```
BUILD CONFIGURATIONS AND BATCH BUILDS

- Each combination of choices made in the build variants user interface is called a build configuration and can be represented textually.
  - For example: “Debug; Target=Linux” is equivalent to these UI settings.

- The build configuration string can be specified as an argument to the model compiler when performing a batch build (e.g. `--buildConfig="Debug; Target=Linux"`).
EVALUATION RESULTS

Build Variants approach was evaluated on $N$ model projects and $K$ target platforms:

- got $K$ times less number of transformation configurations for maintenance (now it is not required to create separate transformation configurations for each platform);
- less maintenance effort when adding new target platform (instead of $N$ new files with new settings we need to update only 2 files with Build Variants declaration and Build Variants implementation);
- managed to decrease the size of SCM repository where models are stored;
- removed inconsistency between end-user builds invoked from the tool and backend builds invoked from automatic testing system;
- reported errors and warning messages during validation of input TCs.
THANK YOU AND JOIN THE HCL TEAM

- DevOps, Agile and Models (MAD @ Models)
  - Industry speakers
- Modeling Tools Restarted – meet HCL