Project Management (Prom)

Deprecated since 0.14

This article is deprecated. The described features are no longer available in current releases.

Prom - Project Management in KIELER

Topics

Overview

The KIELER Compiler (KiCo) can generate different code targets from models. For example it is possible to generate C and Java code from an SCT file. As a result KIELER has to integrate with existing development tools and practices for the C and Java world. In the context of embedded systems, the target device also varies heavily.

Therefore the KIELER Project Management (Prom) has been developed. It eases the creation, compilation and deployment of projects, when using models that can be compiled via KiCo (e.g. SCCharts, Esterel). Furthermore it eases the creation of wrapper code, which is used to deploy, run, or simulate the model.

The features provided by prom include:

- Project and file creation **wizards**
- An incremental **project builder** that performs several tasks, namely
  - Compilation of model files using KiCo
  - Template processing to generate code for deployment or simulation
  - Compilation of simulation code to an executable
- A **DSL for configuration** of the project build

In the following it is explained in further detail how to use and extend these features.

Project Wizards

SCCharts can be compiled for example to C using the KIELER Compiler and there is existing tooling for the C language in Eclipse. Using the SCCharts project wizard, such existing tooling for a target language or platform can be re-used.

Therefore the actual project creation is delegated to another project wizard. Afterwards additional files are created within this newly created project by the SCCharts project wizard. For instance a model file and files for configuration of the build or templates for wrapper code might be added to the project. Further the created project properties are extended with information specific to SCCharts projects, e.g., that the Prom project builder should be used. This approach makes it possible to re-use project wizards from the CDT or JDT and get a working setup with a model file that can be compiled, simulated and deployed with low configuration effort.

Which project wizard from existing tooling should be used and which files should be created afterwards can be configured in the Eclipse preferences. Pre-defined setups for various languages and target platforms can be created this way.

File Wizards

There are various file wizards for the DSL that come with KIELER. These create a file with some default content.

File wizards exist for

- SCCharts text files (**sctx** files)
- Build configurations (**kibuild** files)
- Simulation configurations (**kisim** files)
- Simulation visualization configurations (**kivis** files)
- Freemarker Templates (**ftl** files)

The Project Builder

The incremental project builder is run by Eclipse either in the background when resources changes (*Project* > *Build automatically*), or manually by the user (*Project* > *Build Project*). What and how files are built can be configured using a new DSL (kibuild files). Errors and warnings that occur during the build are added as *markers* to the resources where they occur, which is a known concept in the Eclipse IDE. For instance when working with Java, compiler errors are added as markers to files when they are saved. This is now also possible for SCCharts text files and provides faster compiler feedback to users, e.g. because a model can not be compiled, as long as the automatic build is active.

Several actions are performed when a project is built:

- Model files are compiled
    - Optionally a template is processed for each model to generate the simulation code for the model.
- Simulation code is compiled to an executable, which can be started using the new simulation
- Freemarker templates are processed to generate code.
  Depending of the type of the template, additional variables are injected into the template
    - *Wrapper code templates* are used to create the wrapper code for a specific model.
      Annotations on inputs and outputs in the model can be used to define which code snippets are injected as part of the build. These code snippets typically contain code to read or write the corresponding inputs and outputs.
    - *Simulation code templates* are used to create wrapper code for simulation of models.
      Thus it is a special form of wrapper code template. Instead of user defined annotations, the injected code snippets are determined by the variables in the model.
      This kind of template can be configured as part of a model compiler to automatically generate the simulation for all compiled models.
    - *Simple templates* are self contained and no additional variables are injected.

If all of these are defined, an incremental project build could consist for example of the following steps:

- Build a model file *A.sctx*
    - Afterwards process a simulation template to generate its simulation code *Sim_A.c*
- Compile the simulation code *Sim_A.c* to an executable using gcc
- Create wrapper code for the model, that is ready to be deployed

Note that if the *Build automatically* option is set, it is possible to (re-)start a simulation without the need to (re-)compile the corresponding model beforehand. This is because the simulation executable has been created in the background as part of the build and is updated if the model changes. This results in a faster code-test-workflow compared to the previous approach, in which a model was always re-compiled before its simulation was started.

Build Configuration via KiBuild

The new project builder is configured using a domain specific language, namely KiBuild. Corresponding to the actions that are performed during the build, its configuration consists of *model compilers*, *simulation compilers* and *template processors*. A template processor is either a *simple template processor*, *wr apper code template processor* or *simulation template processor*.

When writing the configuration, use code completion to see available attributes for the entities. The following table describes the available attributes.

Attribute	Domain	Default Value	Description
KiCo Model Compiler			
outputFolder	String	kieler-gen	`The folder in which compilation output is saved`
whitelist	String, Regular expression	-	Only model files that have a location matching this regular expression are compiled. Thus to compile only a specific model, one can use the expression "ModelName.sctx"
blacklist	String, Regular expression	-	Model files that have a location matching this regular expression are exluded from the build. Thus to exclued all models and skip compilation, one can use ".*", which matches everything.
outputFileExtension	String	c	Compiled models are saved with using this file extension. Thus this attribute should match the code format that is generated by KiCo at the end of the compilation.
outputTemplate	String, Project relative file path	-	An optional template to add surrounding code to KiCo generated output for every compiled file. In the template the placeholder **${kico_code}** can be used an will be replaced with the compiled code.

compileChain	String, Id of a pre-defined compilation system or processor id or a project relative file path to a kico file Can also be a list of the above to compile models in several steps Can also be a map to define the compilation of different model types `compileChain { sctx: de.cau.cs.kieler.scchart s.netlist.simple strl: de.cau.cs.kieler.esterel.netlist.simple }`	de.cau.cs.kieler.scchart s.netlist.simple	The compilation system that is used by KiCo to determine the compile chain.
communicateRegisterVariables	Boolean	true	Determines if the variables that save the internal state of a model should be communicated to the simulation generation. If set to false, stepping back and forth in the simulation history will not change the internal state of the model.
Simulation Compiler			
command	String	For C: "gcc -std=c99 -Werror=int-conversion -o \"./${outputFolder}/${executable_name}\"" For Java: "jar cvfe \"../${outputFolder}/${executable_name}\""	The command that is called to compile simulation code to an executable. In case of the C simulation, the compiled file is added implicitly as additional parameter, to create an executable. In case of Java, all class files and the class file of the compiled model are added implicitly to create an executable JAR file.
outputFolder	String, Project relative folder path	kieler-gen/sim/bin	The folder in which compiled output will be saved. Note that it is possible to use a command that creates the compiled files in a different location. However the folder specified in this attribute is created before the command is executed and refreshed afterwards. This ensures that the folder exists and changes will be noticed in the Eclipse workspace.
libFolder	String, Project relative folder path	kieler-gen/sim/lib	The folder where additional files are saved before the command is run. These files can be linked into the simulation code, e.g., for JSON handling.
timeout	int	10	Time in seconds that is waited for the executed command to finish. If the command runs longer, it is assumed to be failed and aborted.
Template Processor			
file	String, Project relative file path	-	The template file that should be processed
target	String, Project relative file path	-	The file in which the output should be saved
Wrapper Code Template Processor			
modelFile	String, Project relative file path	-	The model file that is searched for annotations to determine the code snippets to be injected.
Simulation Code Template Processor			
modelFile	String, Project relative file path	-	The model file that is searched for annotations to determine the code snippets to be injected
compiledModelFile	String, Absolute file system path	-	The absolute path of the compiled model file for which the simulation is created. This is used to replace the placeholder ${compiled_model_loc} in the simulation code template

variables	Map, e.g., `variables { input: myVar1 output: { bool: myVar2 int: myVar3[2][3] } }`	-	Optional additional variables that should be communicated to the outside
interfaceTypes	String, List of Strings	-	The interface types that should be communicated in the simulation, e.g., input, output, internal

Example for KiBuild files:

Simple KiBuild Example

```
// Compile models to C code
model compiler kico {
  outputFolder: kieler-gen   // The folder, in which the compilation output is saved
  outputFileExtension: c     // The file extension for compiled files
  compileChain: de.cau.cs.kieler.scchart.netlist.simple  // The system that determines the compile chain
within the KIELER compiler

  // Generate C simulation for compiled models
  process simulation template {
    file: assets/CSimulation.ftl  // A template for simulation code
  }
}

// Compile simulation code
simulation compiler c {
  libFolder: kieler-gen/sim/lib      // Create additional libraries required for compilation in this folder
  outputFolder: kieler-gen/sim/bin   // Create the executables in this folder
  command: "gcc -std=c99 -o ./${outputFolder}/${executable_name} ${file_path} "  // Use gcc to compile the code
}
```

Complex KiBuild Example

```
// Compile models to Java code
model compiler kico {
  outputFolder: kieler-gen
  outputFileExtension: java
  outputTemplate: assets/OutputTemplate.ftl
  compileChain: de.cau.cs.kieler.scchart s.netlist.simple
  whitelist: "ModelA|ModelB"    // Only compile models that match this regex

  // Generate C simulation for compiled models
  process simulation template {
    file: assets/JavaSimulation.ftl
  }
}

// Compile simulation code
simulation compiler java {
  libFolder: kieler-gen/org/json
  outputFolder: kieler-gen/sim/bin
  command: "javac -cp kieler-gen -d bin \"${file_path}\" "
  jarCommand: "jar cvfe \"./${outputFolder}/${executable_name}\" sim.code.${file_basename} -C bin . "
}

// Process a simple template
process template {
 file: Template.ftl
 target: Output.txt
}

// Process a template to generate a main file that can be deployed.
process wrapper template {
  file: Main.ftl
  target: kieler-gen/Main.c
  modelFile: MyModel.sctx
}

// Process a template to generate a simulation for a model that has been compiled with some other framework
process simulation template {
  file: assets/JavaSimulationForOtherModel.ftl
  target: kieler-gen/Sim_OtherModel.java
  variables: {    // These variables should be communicated to the outside
    input: a,b,c
    output: x,y,z
  }
  interfaceTypes: input, output  // Only communicate these interface types. In this case, internal variables
are not communicated.
}
```

Project Drafts

Project drafts are used to provide default settings for project creation. They are configured in the **preferences** (*Window* > *Preferences* > *KIELER SCCharts* > *Project Drafts*).

An project draft consists of

1. a unique **name**, which may not contain a comma
2. an **associated project wizard**
3. the path of the default **model file** for the project
4. the path of the default **main file** for the project
5. information about **folders and files** that should be imported at project setup

Besides the name, all of these are optional, but can improve the workflow.

The associated project wizard is run as part of the Prom project wizard and takes care of the actual project creation. Afterwards the model file is created and finally other folders and files are imported.

Placeholders

There are some placeholders that can be used in initial resources for projects, which are listed below.

Placeholder	Description
${project_name}	Will be replaced with the name of the project that is created
${modelFile_path}	The project relative path of the initial model file
${modelFile_name}	The name of the initial model file
${modelFile_basename}	The name of the initial model file without file extension

Paths for imported resources

To import a resource (folder or file), its project relative path has to be specified. The resource will be created at this location in the project. Furthermore, it is possible to specify initial content for these resources. This is done in the field *origin*. Without an origin specifed, an empty resource will be created.

To specify intial content for a file, the origin has to be an **absolute file path** or an **URI** with the platform scheme of Eclipse. Such an URI has the form *platform:/plugin/a.plugin.name/folder/in/the/plugin/file.txt* Specifying intial content for a folder is analog. Its origin has to be an **absolute directory path** or an **URI** in the form *platform:/plugin/a.plugin.name/folder/in/the/plugin*

Wrapper Code Generation

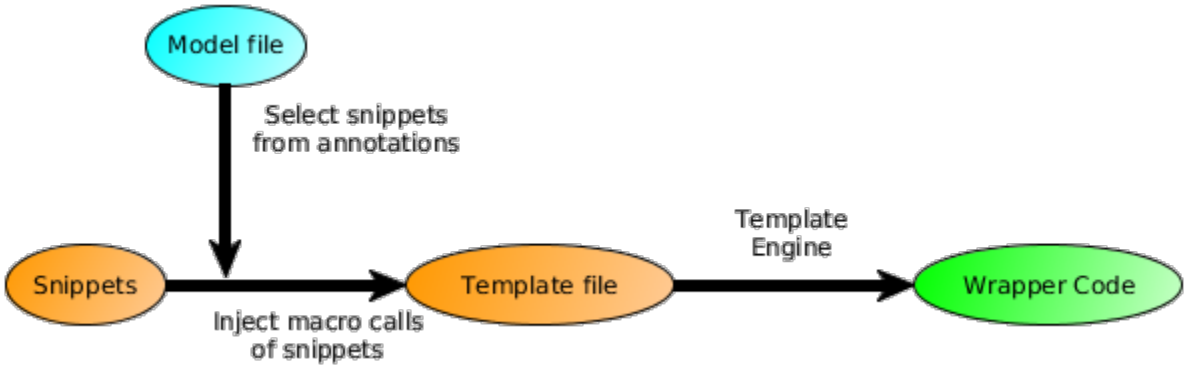
When modeling a program for an embedded system, it is necessary to set inputs and outputs of physical components (sensors/actuators) to inputs and outputs of the model. This is typically done using wrapper code. However, **wrapper code is often similar** for a specific device and programming language.

Therefore one can write **wrapper code snippets** for a target device. These can then be injected to a **template file**. What snippets are injected is defined using **annotations on inputs and outputs** directly in the model file.

In SCT files, annotations are added as in java, with an at-sign e.g. *@Wrapper Clock, "500"*. You can write implicit and explicit wrapper code annotations.

Explicit annotations have the form **`@Wrapper SnippetName, arg1, arg2, ..., argN`**. An explicit wrapper annotation raises an error if the snippet does not exist, thus it is **recommened** to use the explicit **@Wrapper** annotation. Every other annotation is tried as wrapper code annotation as well, but will be ignored, if no such snippet could be found. Thus you can write the above explicit annotation as **`@SnippetName arg1, arg2, ..., argN`**, but there will be no error if the snippet with this name does not exist or could not be found, for example because of a typo.

Note: Annotation **names** and parameters are **case sensitive**. That means that *Clock, clock, Floodlight, FloodLight* are all different annotations.



In the **template file** one can use special **placeholders**.

${file_name} is replaced with the name withouth extension of the file that is generated (e.g. *Main.java* will be *Main*).

${model_name} is replaced with the name of the last compiled model.

${declarations}** and **${decls} will be replaced with additional declarations of variables and functions (<@decl>...</@decl> of a snippet definition). Declarations should occur before the tick loop of the model file. In general they are not required for Java code but may be useful in C applications (e.g. for *extern* calls).

${initializations}** and **${inits} will be replaced with initialization code for components (<@init>...</@init> of a snippet definition). Initialization should occur before the tick loop of the model file.

${inputs} will be replaced with code to set inputs for the model (<@input>...</@input> of a snippet definition). Setting model inputs should occur in the tick loop, before the tick function call.

${outputs} will be replaced with code to read outputs of the model. (<@output>...</@output> of a snippet definition). Reading outputs of the model should occur in the tick loop, after the tick function call.

${releases} will be replaced with code to free allocated resources. (<@release>...</@release> of a snippet definition). Releasing resources should occur after the tick loop at the end of the program.

```
...

// Initialization
${inits}
...
 // Tick loop
while(...){

    // Input snippets
    ${inputs}

    // Reaction of model
    scchart.tick();

    // Output snippets
    ${outputs}
}
...
```

To ease the modification of the template file, one can open it with the text editor the final code will be for. This will enable syntax highlighting and code completion for the langauge, but it will not show any errors. You can open the file for example with the Java Editor of Eclipse using *Right Click > Open With > Other > Java Editor*

Simulation templates

The task of the simulation code is to read the inputs from the KIELER user for the simulation, execute a tick, then send the outputs that have been produced back to KIELER. The communication with KIELER is done using a JSON format.

To create the simulation code, a template is used in which code is injected for each variable in the model to fill the JSON object with the current variable values. This way, the state of the model is communicated to the outside. Before the tick, inputs can be set in the model. Thus there is also code injected for each variable in the model to change its value using a JSON input.

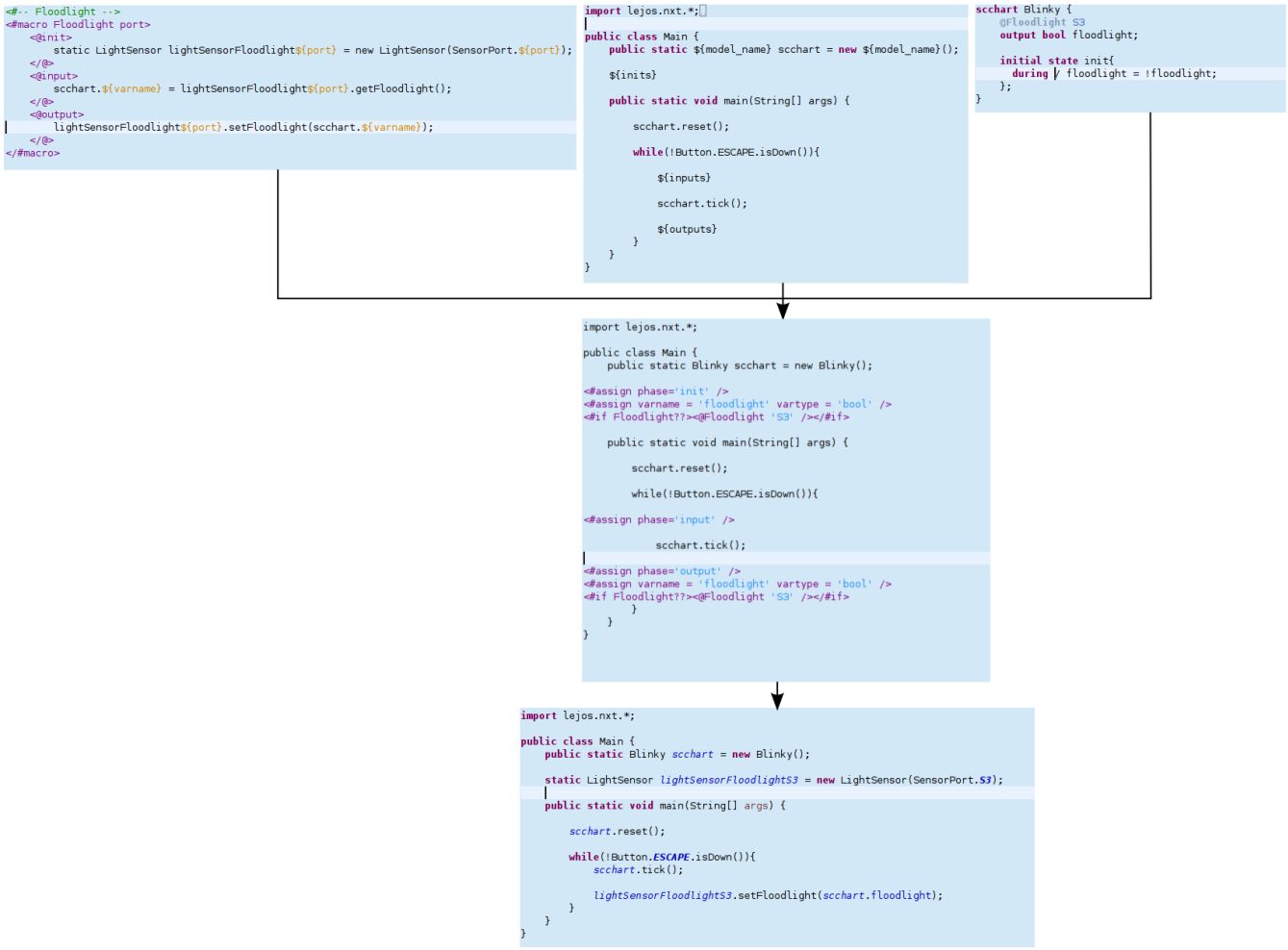
In conclusion, the simulation code generation is a special form of wrapper code generation. For a simulation template, the injected code snippets are not selected from annotations in the model. Instead code is injected in a specified form for all variables to communicate their states using a JSON format.

FreeMarker

The wrapper code injection is done using the open source **template engine FreeMarker**. A wrapper code snippet is basically a Macro definition of FreeMarker. The Macro is called when the corresponding annotation is found in the model file. The file extension of FreeMarker templates is **.ftl**.

There is an Eclipse plugin for FreeMarker as part of the JBoss Tools Project. It can be installed using the Eclipse Marketplace.

Example for wrapper code generation from annotations:



Problem Solving

CDT Projects

When working with the CDT, the folder that contains the simulation code has to be excluded from the CDT build, because this code is compiled using the compiler specified in the kibuild file, and every simulation file has an additional main function, which is not the use-case that a CDT project is made for. These files are self contained and do not interact with other files in the CDT project, thus they should not be built.