Comfortable SCCharts Modeling for Embedded Systems

Andreas Achim Stange

Bachelor Thesis
2015

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems group
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Steven Smyth

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

The Eclipse based KIELER development environment provides tools to create and compile SCCharts, which is a visual language to model embedded systems. The workflow for SCCharts in KIELER contains steps that can be automated. This thesis discusses a solution design that allows the user to focus on the model to be developed by removing the manual processing of these steps. It introduces an easy setup and execution of SCCharts projects.

The implementation includes a project creation dialog, which adds files to a project so that it is directly executable. The project execution itself compiles SCChart model files, assembles wrapper code for these, and deploys the result via a list of user defined shell commands. Furthermore, a container for default settings to create and execute projects for different target devices and operating systems is introduced.

After a comparison of open-source template engines for Java, the FreeMarker engine has been identified as suitable for wrapper code generation as part of launching an SCCharts project. Finally, the results of a short survey are presented, which show that the implemented tools enhance the workflow when using SCCharts in KIELER to model embedded systems.

Contents

List of Figures

Listings

List of Tables

Abbreviations

KIELER	Kiel Integrated Environment for Layout Eclipse RichClient
MoC	Model of Computation
SCT	Textual SCCharts Language
KiCo	KIELER Compiler
leJOS	LEGO Java Operating System
NXC	Not Exactly C
CDT	C/C++ Development Toolkit
UI	User Interface
GUI	Graphical User Interface
JS	JavaScript
JSP	Java ServerPages
JAR	Java Archive
API	Application Programming Interface

Chapter 1

Introduction

Today we live in a world where embedded systems surround us. Little computers and microchips are built into everyday utilities such as mobile phones, heaters and credit cards. With sensors and actuators they can react to and influence their environment and many embedded systems are part of safety-critical systems e.g. in cars and avionics. This drastically increases the requirements of correctness, reliability and maintainability. At the same time, they should be easy to use without burdening the user with their complexity.

To meet these high requirements specialized languages to design, implement and simulate embedded systems have been developed. This includes graphical modeling tools such as SyncCharts [And96] and Ptolemy II [Pto14] as well as textual programming languages such as Esterel [BC84] and Lustre [HCR+91]. They emphasize determinism of concurrent execution with respect to correctness and timing.

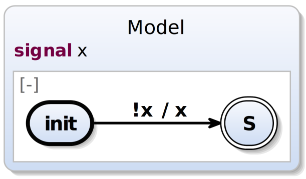
A major advantage of graphical languages is the visual representation of dependencies, control flow and states, which eases the verification of models by the human eye. However, this advantage often comes with additional work to layout and edit the graphical model. In contrast to this, when working with a text editor, one can add and remove letters anywhere without manually moving subsequent text to make place. For large models, finding a visual representation that is understandable and maintainable can be very time consuming and distracts from the development of the system itself [FH10]. Thus meeting the high requirements of modern embedded systems is not only challenging but also tedious when working with time consuming large visual models or tools that are not specialized for this kind of development.

1.1 SCCharts

SCCharts is a visual language designed for safety-critical systems [HDM+13]. It bases on SyncCharts notation. However, semantically it uses a Sequentially Constructive Model of Computation (MoC), which is a superset of the synchronous MoC used in SyncCharts. This gives the developer a greater freedom as some common programming patterns can be expressed in SCCharts although they are not valid in SyncCharts. Figure 1.1 illustrates this by using a sequential access pattern in SCCharts that is known from imperative programming.

```
x=false; if(!x){x = true;...}
```

(a) Sequential access pattern



(b) Sequential access in SCCharts

Figure 1.1. Example of sequential read and write access to a variable in SCCharts. The model will start in the initial state `init`. When `x` is absent the state is changed and `x` is emitted, making it present. Afterwards, the program finishes in the final state `S`. This is not possible in SyncCharts semantics.

However, by extending the synchronous MoC, SCCharts inherits its deterministic basis. A program behavior is a sequence of reactions, each consisting of reading the current inputs, calculating outputs and updating its internal state [HCR+91]. The reaction of a program corresponds to a transition of an automaton and—conceptually—does not take any time. Thus it is considered atomic, although finitely many processor operations may be executed as part of it. A reaction is called a *tick*. Ticks are used as abstract measurement of time, independent of the physical time.

SCCharts provide a range of elements to construct programs. However, most of these are additional tools for the developer and can be expressed using a set of other elements. Therefore a distinction is made between *Core SCCharts* and *Extended SCCharts* which are shown in Figure 1.2. As a result, the low-level side of a compiler for SCCharts only needs to handle Core elements. Other elements can be transformed to Core SCCharts on a higher level. This eases the work of compiler developers.

The visual notation of SCCharts emphasizes readability and eases the understanding and verification of models whereas the extension of the synchronous MoC gives the developer a greater freedom and a sound basis. Furthermore, the identification of Core SCCharts eases the development of a compiler for the language. In conclusion SCCharts tries to ease the process of modeling and implementing embedded systems.

1.2 KIELER

The Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) is a project of the Real-Time and Embedded Systems group at Kiel University.[1] It builds on top of the Eclipse plugin architecture and frameworks. One of the main goals of the project is to ease the development of graphical models for embedded systems by providing an automatic layout that tightly integrates with the framework. With KIELER it is possible to

[1] http://www.rtsys.informatik.uni-kiel.de/en/research/kieler

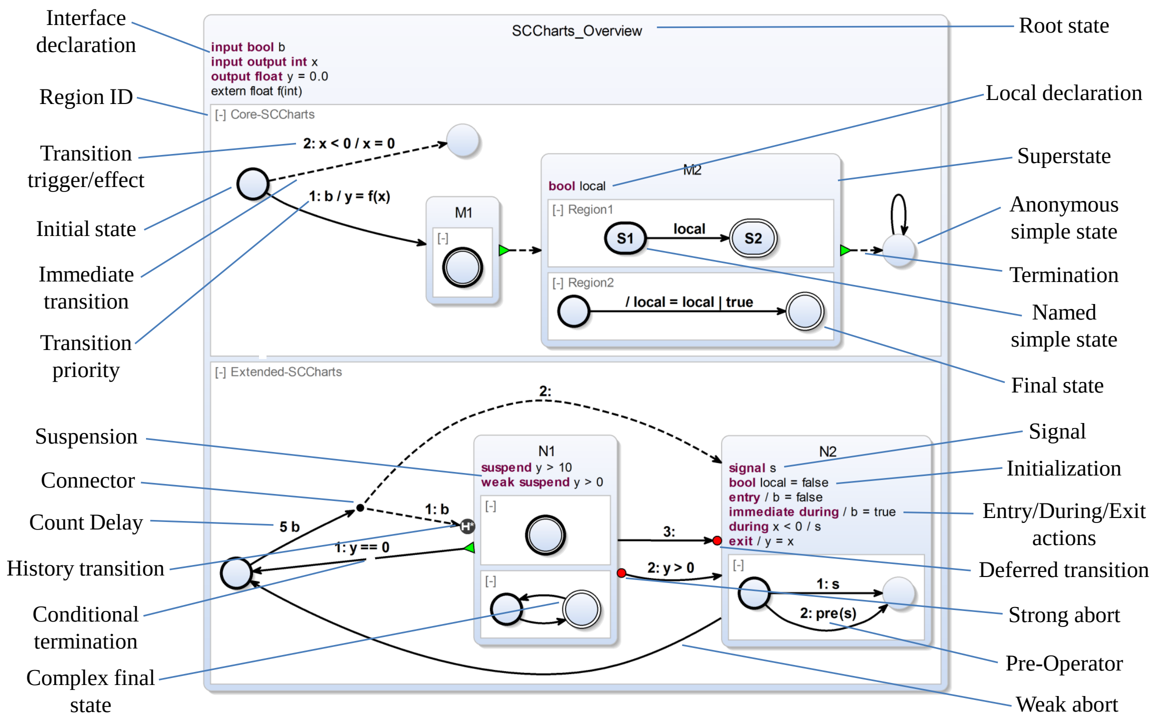
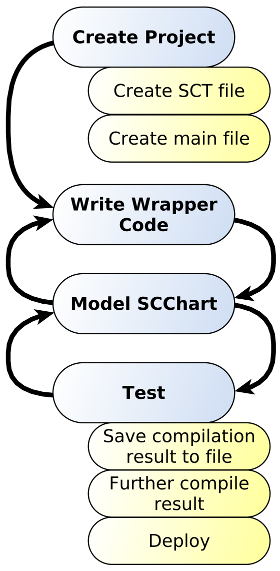
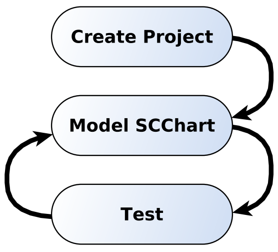


Figure 1.2. The image illustrates the elements of SCCharts [HDM+13]. The upper half shows Core SCCharts whereas the lower shows extended SCCharts.

edit, browse and visualize models. The layout algorithms are an actively developed aspect of KIELER and also used in other projects. The other main working area of the Real-Time and Embedded Systems group is the semantics of modeling languages in general and SCCharts in particular. Using KIELER, SCCharts can be created, visualized and compiled to different programming languages, including C and Java [MSH14]. SCCharts in KIELER are not created with a palette of possible elements that can be assembled using a graphical interface, although this is a common approach for editing graphical models. SCCharts are modeled in the Textual SCCharts Language (SCT), which is then rendered and laid out by aforesaid algorithms. This avoids typical drawbacks of modeling with a graphical palette.



(a) Current workflow



(b) Desired workflow

Figure 1.3. Comparison of the current and the desired workflow. The project setup should create all needed files automatically and wrapper code should be generated from information provided in the model. To test/launch a project, the manual compilation and deployment should be unnecessary.

1.3 Problem Description

Although the KIELER tools provide new ways to ease the graphical design of models, there is not yet a streamlined workflow to develop applications using SCCharts. This is apparent in the fact that the user needs to manually compile SCT files to the desired target language and afterwards save the output—as well manually—to the desired file path. As this is a recurring activity, it prevents the user from a short code-test workflow.

Another required task to create a project that can be executed is to write wrapper code for the compiled SCChart and to set inputs and outputs to correct components. Although the user should be able to focus on the model in development, the wrapper code creation consumes time and thought and distracts from this task. Furthermore, wrapper code is dependent on the target language and thus cannot benefit from the different SCCharts compilation targets supported by KIELER.

The goal of this thesis is to ease the creation, initialization and execution of SCCharts projects. This includes the creation of Eclipse plugins which provides dialogs to create and initialize SCT files, SCCharts projects, and launch configurations.

The project creation will be compatible with and utilize existing dialogs and settings of the currently installed Eclipse. For example it will be possible to create an SCChart project for Java or C if support for these are installed in Eclipse. A newly created project should be ready to be executed. Therefore, it has to be initialized with at least one file for the model and a file with wrapper code. The wrapper code is used to initialize and run the model.

Launching a project will compile SCT files, generate wrapper code from these files and afterwards further compile and deploy the output to the target platform. As SCCharts is a general purpose language, the project launch has to be generic, such that arbitrary target platforms and languages for wrapper code can be used. Figure 1.3a illustrates the current workflow whereas Figure 1.3b shows the described workflow in KIELER.

To keep the manual configuration that users have to perform at a minimum, a dialog to define default settings has been implemented as part of this thesis. Further, initial content for a demonstrator device will be shipped with the developed plugin.

Aforesaid integrates with common workflow and practices as well as plugins of the Eclipse environment.

1.4 Thesis Outline

Chapter 2 will introduce related work and tools which can be used to model and implement embedded systems in general, whereas Chapter 3 covers technologies used to implement the created plugins of this thesis. Further, an embedded system and corresponding firmware is presented. It has been used to test the resulting tools.

Chapter 4 introduces the solution design for aforesaid problems. It includes an overview and evaluation of open-source template engines for Java, which can be used to create wrapper code within the KIELER tools.

Chapter 5 gives a top-down view of the implementation, starting with an overview of used Eclipse plug-ins and package dependencies, to a more detailed illustration of associations between classes and how they interact.

Chapter 6 contains a brief comparison and evaluation of the old and new workflow in KIELER when developing SCCharts. Finally, Chapter 7 concludes this thesis and discusses possible future work.

Chapter 2

Related Work

The following chapter presents tools related to developing embedded systems. They aim at providing a concise workflow for application development. The first introduced tool is the C/C++ Development Toolkit. Afterwards the SCADE Suite is presented.

2.1 C/C++ Development Toolkit

The C/C++ Development Toolkit (CDT) is a collection of plugins for the Eclipse platform.[1] They provide an IDE for C and C++ in general and thus can be used to program embedded systems in particular.

The CDT provides typical IDE features including

▻ project creation

▻ project build via *make*

▻ source navigation

▻ syntax highlighting

▻ auto-completion

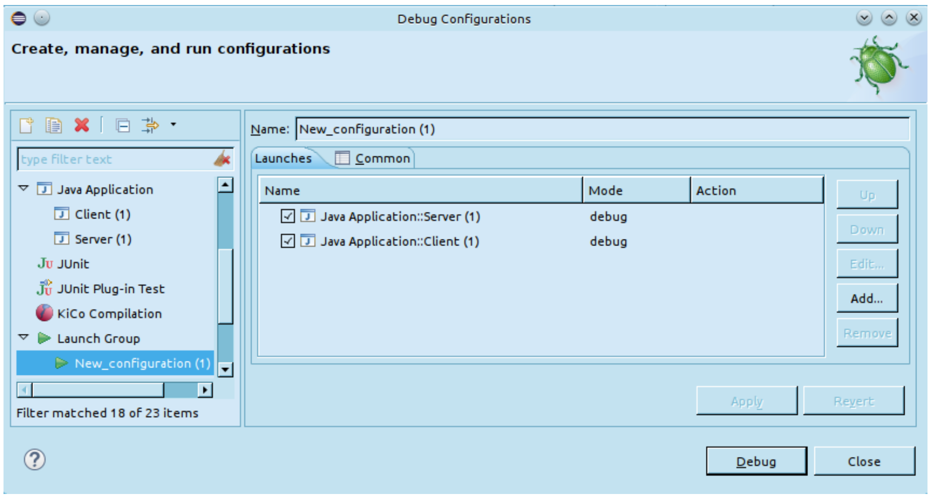
▻ refactoring

▻ debugging tools

Furthermore, it is possible to add arbitrary launch compositions via so called *launch groups*. These can be used e.g. to compile C to binary code and afterwards deploy the output to a target system. Therefore launch groups contain a list of Eclipse launches which can perform arbitrary tasks, including the execution of shell commands. Thus it is possible to compile, deploy and run a project with a single user action if the IDE is configured accordingly.

The CDT can be used to create software for a wide range of embedded systems. This is achieved with the option to use several toolchains. A toolchain is a set of programming

[1] `https://eclipse.org/cdt/`



Name: New_configuration (1)

Launches | Common

Name	Mode	Action
☑ Java Application::Server (1)	debug	
☑ Java Application::Client (1)	debug	

Figure 2.1. Screenshot of a launch group for a client-server application. The server is started before the client.

tools which are typically executed sequentially, each with the output of the respective predecessor in the chain. In the context of embedded systems, toolchains typically consist of compiler, linker and debugger. They will most likely be used to cross-compile a project, which means to compile for a different processor architecture than it is developed on. For example a PC running Linux with an x86 architecture can be used to write code that will be deployed to an ARM micro-controller with the corresponding toolchain.

There are additional extensions to the CDT to target embedded systems such as the GNU ARM Eclipse project.[2] The project provides support of toolchains for the ARM processor family, as well as simulation and debugging of applications using third party projects. In addition there are C/C++ project templates. These generate projects, which can be executed without further modification.[3]

In conclusion the CDT can provide a workflow from project setup to compilation and deployment for embedded systems.

2.2 SCADE Suite

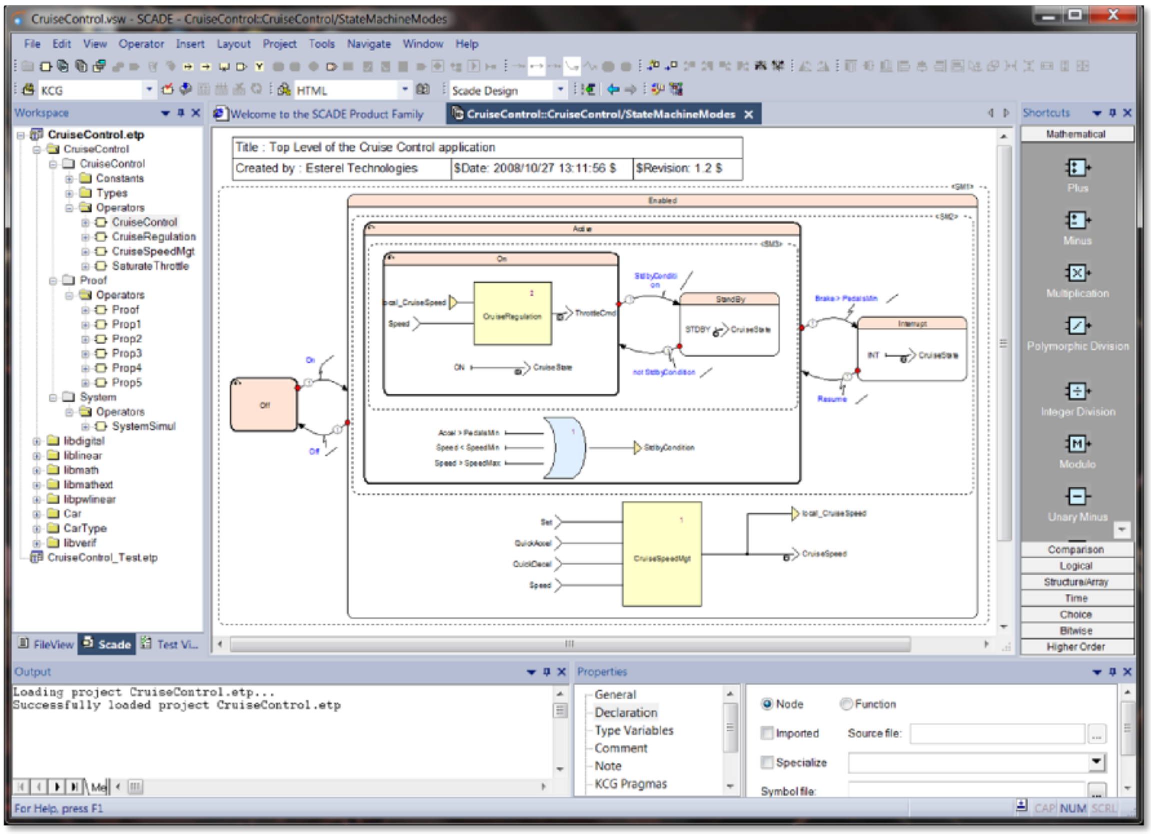
SCADE Suite is a development environment for safety-critical embedded software systems by Esterel-Technologies.[5] It integrates with the Scade language, which is based on the synchronous data-flow language Lustre.

[2] http://sourceforge.net/projects/gnuarmeclipse/

[3] http://gnuarmeclipse.livius.net/blog/templates/

[4] http://www.esterel-technologies.com/wp-content/uploads/2015/03/SCADE-Suite-IDE-View.png

[5] http://www.esterel-technologies.com/products/scade-suite/



```
Loading project CruiseControl.etp...
Successfully loaded project CruiseControl.etp
```

Figure 2.2. Screenshot of SCADE Suite showing a graphical model in the center surrounded by typical IDE windows.[4]

The SCADE Suite has tools for

▻ model-based design

▻ documentation generation

▻ simulation of models

▻ verification of models

▻ prototyping

The compiler can produce C and Ada code and has been certified for high standards in Avionics (DO-178B&C), Railways (EN 50128) and Automotive (ISO 26262). In combination with the formal verification of programs via model-checking this can drastically reduce certification costs.

SCADE Suite is an industry standard all-in-one solution for embedded system design which has been used and proven in many critical software applications ranging from avionics over railways to automotive and nuclear power plants.[6]

[6] http://www.esterel-technologies.com/success-stories/

Chapter 3

Used Technologies

The following chapter names technologies used to implement and test the problem solution. First we take a deeper look on the Eclipse platform and its extension mechanisms as KIELER is based on them. Afterwards we take a look at the programming language Xtend, which has been used to program the solution. Finally we take a look at a simple embedded system that is used as demonstrator for a target device.

3.1 Eclipse

The Eclipse project is an open source platform originally developed as Java IDE by IBM.[1] It is a large and actively developed project, maintained by the Eclipse Foundation.

Eclipse runs on Windows, Mac and Linux and although it has been started as Java IDE, its powerful extension mechanisms have grown a wide range of other tools for the platform. Today Eclipse has over hundred extensions with modeling tools, programming languages and frameworks.[2] Many of the tools are developed to be used by other developers. New plugins integrate seamless with the platform. However, it is also possible to ship a standalone application for end-users.

The Graphical User Interface (GUI) of Eclipse uses the Standard Widget Toolkit, which itself uses native GUI elements of the supported operating systems.

3.1.1 Plugin Architecture

Eclipse can be easily extended, which is part of the platform's success. This is achieved by a plugin mechanism that bases on *Extensions* and *Extension Points*.

Extension Points are used to define interfaces which afterwards can be implemented by an arbitrary number of Extensions. An interface consists of named fields with a specified domain. In Eclipse version 4, the domain can be a string, a boolean value, an identifier, a resource, or a fully qualified name of a Java class. Extension Points and Extensions are both described using XML.

[1] https://eclipse.org/eclipse/

[2] https://projects.eclipse.org/list-of-projects

Extensions are implementations for Extension Points. They have to provide values for the named fields. With Extensions, new functionality is added to the platform and they can be contributed by any plugin.

For example there is a plugin that provides an Extension Point for context menu items. It defines fields for a unique identifier of the implementing Extension, a label for the menu item, an identifier for the context menu the item should be added to, and a Java class that defines what will happen if the item is clicked. This Extension Point has been implemented by several plugins, giving the platform new functionality.

The plugin mechanism enforces a modular and incremental development of projects which are key principles of software engineering. Moreover Eclipse is automatically handling dependencies of plugins and has an update manager, which simplifies installing and maintaining a productive working environment.

3.2 Xtend

Xtend is a Java dialect that compiles to Java code.[3] It integrates seamless in Eclipse as it builds upon frameworks which have their basis on the Eclipse platform.

Xtend is a powerful, modern language that gives the developer a way to write more compact code than with pure Java. Nonetheless Xtend is fully compatible to existing Java classes and frameworks. Thus it is possible to use Xtend classes in Java and vice versa.

Features of Xtend that are not present in Java 7 include[4]

Extension methods
It is possible to add methods to existing classes in Xtend.

Lambda expressions
Xtend has a very compact syntax for unnamed functions.

Switch-expression for any object
A switch expression in Xtend can compare any objects and has support for automatic casts as illustrated in Listing 3.1.

Implicit casts
After the usage of `instanceof`, a cast is done implicitly.

Properties
In Xtend getters and setters are called implicitly.

[3] http://www.eclipse.org/xtend/
[4] http://www.eclipse.org/xtend/documentation/index.html

```
def length(Object x) {
   switch x {
      String case x.length > 0 : x.length
      List<?> : x.size
      default : -1
   }
}
```

Listing 3.1. The example utilizes the automatic cast of switch-expressions in Xtend to return either the length of a string, or the size of a list. If the object does not match any of the types, the default value `-1` is returned.[5]

```
final List<String> list =
   new List<String>();
list.add("a");
list.add("b");
list.add("c");
```

(a) Java code that creates a list

```
val list = #["a", "b", "c"]
```

(b) Xtend code that creates a list

Listing 3.2. The Listing 3.2a shows Java code whereas Listing 3.2b shows equivalent Xtend code. The Java code is obviously longer than the Xtend code. This is due to type inference, syntactic sugar for lists and **`val`** as short expression for a final variable.

Literals for collections
An unmodifiable list, map, set and array can be created using short syntax.

Type inference
Xtend code can be shorter than Java code because redundant type information is inferred.

Semicolons are optional
A semicolon after every statement, which is needed in Java, is optional in Xtend.

Furthermore Xtend is a functional language in the sense that everything is an expression with a return type. The syntax is often more compact compared to equivalent Java code which is obvious in Listing 3.2.

Xtend has been used in the KIELER project and is used for the implementation of this thesis.

[5]Example taken from the Xtend documentation.

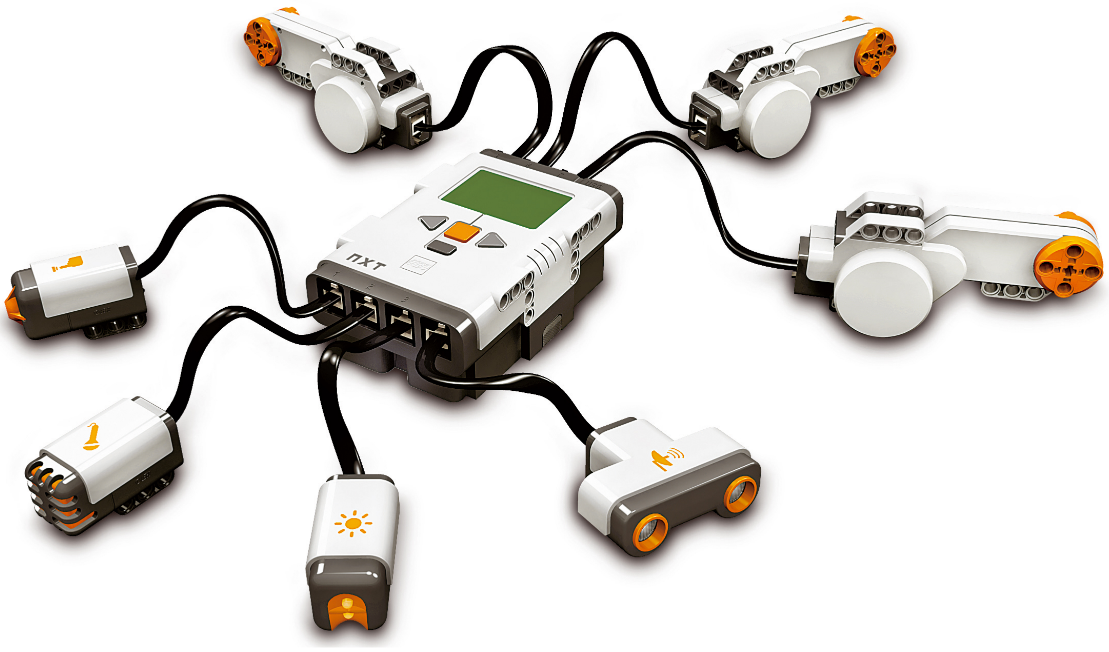


Figure 3.1. Photo of an NXT brick with sensors and motors.[8]

3.3 Mindstorms

Mindstorms is a product family from Lego, with sensors, motors, and a programmable brick.[6] The newest iteration of the product family is the EV3 programmable brick. Its predecessors are NXT and RCX. Although the solution design of this thesis is applicable for all Mindstorms, it focuses on the NXT brick and related software.

Sensors for the NXT brick include light, touch, ultrasonic, and sound sensors. The brick has 3 ports to connect actuators (port A-C) and 4 ports for sensors (port 1-4).

Although there is an operating system as well as development tools delivered with the NXT brick, these are purely graphical tools, targeted at programming novices. Therefore several replacements for the firmware have been developed, supporting many well known programming languages such as Java, C/C++, Python, and Lua.[7]

The Mindstorms NXT brick will be used to demonstrate embedded system design via SCCharts and at the time of writing, the KIELER Compiler (KiCo) can generate Java and C code from these. Therefore we first take a closer look at a Java Virtual Machine (JVM) for the brick. Afterwards we take a look at a C dialect for Mindstorms.

[6] http://www.lego.com/en-us/mindstorms/

[7] http://www.teamhassenplug.org/NXT/NXTSoftware.html

[8] https://jaxenter.de/wp-content/uploads/2013/08/loewenstein_1.jpg

```
#define PAUSE 100

task saySomething() {
   while(true) {
      TextOut(1, 1, "0");
      Wait(PAUSE);
   }
}
```

```
task saySomethingElse() {
   while(true) {
      TextOut(1, 1, "1");
      Wait(PAUSE);
   }
}

task main(){
   TextOut(1, 1, "hello world!");
   Precedes(saySomething,
       saySomethingElse);
}
```

Listing 3.3. Example showing an NXC program that prints `hello world` and afterwards starts two concurrent functions.

3.3.1 leJOS NXT

The LEGO Java Operating System (leJOS) is an open source replacement for the Mindstorms firmware.[9] It equips the brick with a small JVM and provides a Java Application Programming Interface (API) for all standard sensors and actuators of the NXT brick. Thus, it is also called *NXJ*.

The leJOS ships with several command line programs that are available for Linux, Mac and Windows. These are used to compile and deploy programs. Additionally there are command line tools to browse files and to fetch logging output of the brick.

Furthermore, there is an Eclipse plugin for NXJ that contribute dialogs for project setup and execution.

This, in combination with the good documentation of the API, makes leJOS a good choice for general purpose programming of NXT bricks for experienced Java programmers as well as beginners.

3.3.2 Not Exactly C

Not Exactly C (NXC) is a C dialect that can be compiled to the NXT compatible assembler language *Next Byte Code*.[10] The compiler is freely available for Linux, Mac and Windows.

In contrast to C, the file extension of the dialect is *.nxc* and it is not possible to include system libraries with an angle bracket syntax. Moreover, the entry point for a program is not a function with name *main*, but a *task* with this name. An example for this is shown

[9] `http://www.lejos.org`

[10] `http://bricxcc.sourceforge.net/nbc`

in Listing 3.3. Further differences of the languages are not discussed here as they are irrelevant for this thesis.

A task in NXC is similar to a Java thread as both are run concurrently. The main task is executed automatically when the program is started whereas all other tasks have to be called explicitly. On the NXT brick, there can be up to 255 tasks executed at the same time.[11]

As C is an imperative language and Java object oriented, their API for the NXT brick naturally diverge. This makes leJOS and NXC a good choice to test the applicability and usability of the tools developed in this thesis.

[11] http://www.hsu-hh.de/download-1.5.1.php?brick_id=CRNFMznv6bTsekow

Chapter 4

Solution Design

The Eclipse platform provides a variety of Extension Points and API, which gives developers a great freedom how to implement a certain feature. Furthermore, there are several possibilities to ease wrapper code generation.

This chapter firstly gives a solution to compile model files, assemble wrapper code, and deploy the output via launch configurations. Secondly a default container for an easier setup of these configurations are introduced and thirdly it is explained how to create and initialize a project by utilizing already installed project wizards and afterwards adding files to the newly created project. Lastly open-source Java template engines are compared and it is explained how the FreeMarker engine can be used to assemble wrapper code.

4.1 Compilation with the KIELER Compiler

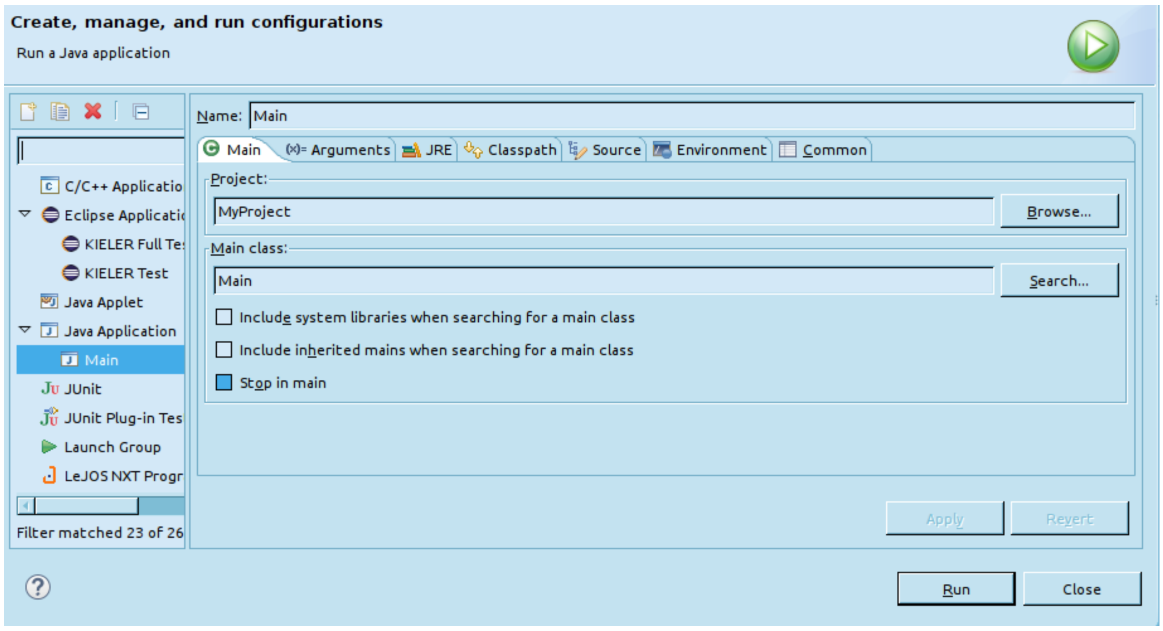
Eclipse provides a mechanism to compile files in a background job. This is done via *builders*. For example, there is a builder registered for Java projects. It compiles Java files to their binary class file representation as soon as they are saved. If compilation errors occur, they are highlighted in the corresponding Java file. This build is done implicitly and is independent of a project's launch.

The builder mechanism works with differences of files. If a file is changed, the difference to its previous version is computed. This is the file's *delta*, which is passed as argument to the builder. However, the KIELER Compiler does not work on a delta basis. It needs to recompile the whole file. Furthermore, compiling larger models is time intensive. As result the builder mechanism of Eclipse is not an optimal solution to compile models with KiCo.

Additionally, in most cases the KiCo generated output has to be further compiled for the target system (e.g., the NXT brick). For leJOS this is done with command line tools, which are used as well to deploy and optionally run the output.

Code generation via KiCo and the further compilation and deployment can be combined in a single project launch. This feature can be implemented using a *launch configuration* (*launch config*) Extension Point.

The User Interface (UI) for such a launch config is presented in a group of tabs. For example the launch configuration for Java applications has a main tab to specify the project and main class it is used for. This can be seen in Figure 4.1. On another tab the user can



Name: Main

Project: MyProject

Main class: Main

☐ Include system libraries when searching for a main class

☐ Include inherited mains when searching for a main class

■ Stop in main

Figure 4.1. Launch configuration for a Java project

specify arguments that should be passed to the program and further tabs are used to define a JVM and execution environment.

To launch a KiCo project several information is needed:

- ▻ The project for which the configuration is used
- ▻ SCT files which should be compiled
- ▻ The target language for KiCo compilation
- ▻ Commands used to further compile and deploy the output

As there might be multiple SCCharts in a project that perform independent computations, a list of model files is a better solution than an input for a single file.

The target languages supported by KiCo can be extended with plugins but do not change most of the time and are limited. Thus, a combobox that fetches possible target languages from KiCo is a suited control to communicate this setting.

The commands used to further compile and deploy the project depend completely on the target system and compile chain. As a result, the user should be able to define a list of commands that are executed sequentially. If a command exits with an error, following commands will not be executed. To illustrate, if the compilation fails it does not make sense to execute the following command for deployment.

The launch configurations for Java as well as for C/C++ have a main tab to specify their project. For a concise UI, it makes sense for the KiCo launch config to present the project selection on such a tab as well. Another tab should be filled with controls regarding code generation by KiCo. This includes the list of model files and the target language combobox. Finally, the third tab contains controls to specify the list of commands.

Most commands will depend on the project structure. For instance, the command to compile a file depends on the file path. Thus, renaming a file would break a launch configuration if the command does not change. In conclusion it makes sense to create a placeholder for the file path, that is required in all projects and used in most commands. This is the main file, containing the entry point of the application. The main tab can contain a text input to specify the main file such that references to it in other text fields can be done using placeholders.

The Eclipse platform provides API and a *variable selection dialog* to add and use such placeholders. This placeholder mechanism is also used in other Eclipse launch configurations, including *External Tools*, and *Eclipse Application*.

With the described launch configuration it is possible to compile an SCChart via KiCo and deploy the result to a target device if a main file with corresponding wrapper code is present.

Wrapper code for a specific device and OS is often similar. Mapping sensor values to inputs of a model is done with code that differs only in some parameters such as the attached port of a sensor. The same applies to mapping outputs of a model to actuators of the device. To ease this mapping process, the described launch configuration will be extended to configure a wrapper code template and definitions of wrapper code snippets. As part of a project launch, needed snippets will be added to the template file, such that wrapper code for the current model is generated.

SCCharts has a feature to annotate elements e.g. states and variables. The syntax for annotations is an at-sign followed by a name and optionally a list of parameters. These annotations can be used, for example on an input variable, to define which wrapper code snippet is appropriate. To illustrate, a variable `input int lightValue` can be annotated with `@LightSensor S3`. This could mean, that a light sensor, that is attached to the port S3, should be mapped to the given input variable.

The described wrapper code generation assembles code snippets and adds them to a template file. In contrast to this, KiCo takes a model file to create semantically equivalent code in a given language. These are fundamentally different procedures. As a result the wrapper code generation should not be implemented as part of KiCo.

To implement the described wrapper code generation, it is necessary to have a syntax that can be used to define parameterized snippets such that parameters are replaced at runtime with values from the model. A *template engine* is a tool to generate text output by defining templates with placeholders, which are later replaced with appropriate values. In conclusion a template engine is a suited tool for the described wrapper code generation.

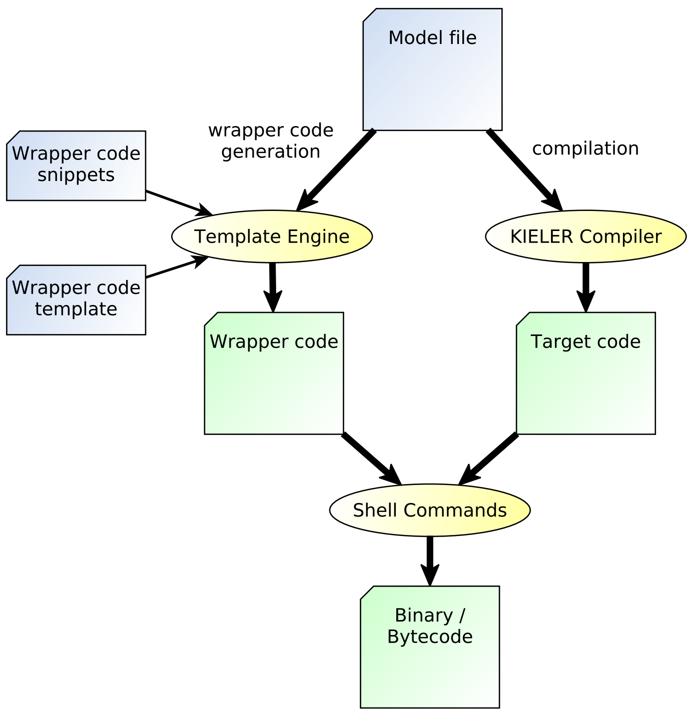


Figure 4.2. Illustration of the steps performed when launching an SCCharts project

Figure 4.2 illustrates the steps of a project launch with KiCo. It is obvious that the wrapper code generation and compilation by KiCo can be done at the same time because they don't interact with each other.

The steps of such a launch configuration may raise unexpected errors. For example there might be a syntax error in the model file such that KiCo is not able to compile it. Another problem occurs if snippet definitions or the commands to be executed could not be found because there is a typographical error in a file path.

In Eclipse, errors from a launch are presented to the user in a Console View. Thus all errors raised as part of the new launch config will be caught and displayed in such a view. Therefore the complete launch process will be wrapped in a try-catch block. This is efficient because it covers all possible errors and does not alter error messages.

To prevent unnecessary configuration labor, all generated output will be saved in a directory *kieler-gen* by convention. Anyhow, the folder structure of source files will be retained. For instance, compiling a file *src/models/subdirectory/MyModel.sct* will result in an output file *kieler-gen/models/subdirectory/MyModel.java*.

4.2 Default Settings for Different Target Platforms

A launch configuration as described above may contain a lot of information. If the user needs to set this for every project manually, it becomes a tedious task.

Eclipse provides an Extension Point for *launch shortcuts*. These can be used to directly create and run a launch configuration. Needed information is gathered from the current state of the IDE and the file to be run.

However, the KiCo launch configuration requires data that can not be inferred just from the model file a launch shortcut may be used on. The target language for the KIELER Compiler, the list of commands and additional data for wrapper code generation needs to be set manually.

This data as well as data to set up a project depends on the target device a developer wants to deploy to. Thus, it makes sense to prepare a container for default settings to be used for a specific target device and operating system, e.g., the NXT brick with leJOS. This is the target *environment*.

In contrast to project specific settings or file properties, environments contain global data. They provide defaults to create and launch projects within KIELER. In Eclipse, global settings are stored and configured in the preferences.

Although environments provide default settings to launch a project, they should not dictate them. Thus it must be possible to alter every setting for a project individually. For launch configurations this can be done using the corresponding dialog. Anyhow, for convenience it makes sense to provide a button that will revert the values to the settings of an environment. This can easily be implemented and makes errors from manual editing reversible.

4.3 Project Creation Dialog

For a good workflow the project setup needs to be as simple as launching a project. At the time of writing there is no project creation dialog (project wizard) dedicated to KIELER SCCharts. This does make sense as SCCharts are meant to be used together with a target language such as C or Java, which have project wizards on their own.

However, when using SCCharts it is desirable to have a wizard that creates a runnable project structure for a given target device. It should be ready to be executed with aforesaid launch configuration. Therefore, a project for SCCharts requires an SCT file with at least one initial state as well as a file with wrapper code to initialize and run the model. Additionally, wrapper code snippets may be imported as part of the project setup.

Figure 4.3 illustrates the behavior of a project wizard that is described in the following. Initializing a project for SCCharts can be implemented as extension of another project wizard because the required files can be added to any project. However, it is not possible to

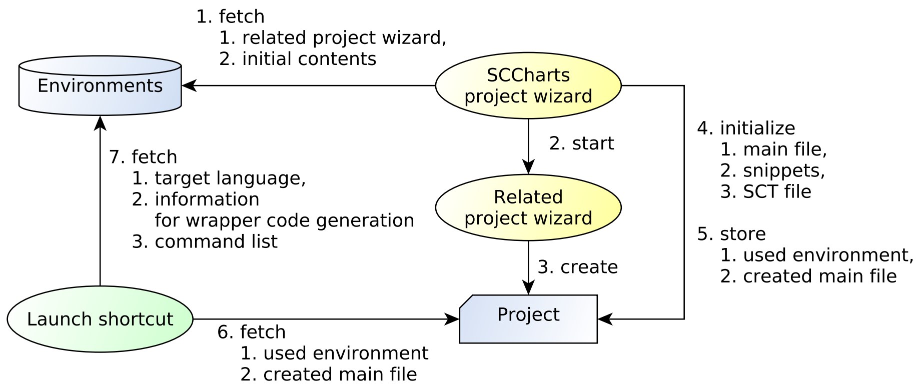


Figure 4.3. Information flow from environments to project creation and project launch

directly extend a wizard in Eclipse. As workaround it is possible to use a wizard as part of another wizard. Thus a project creation dialog for SCCharts can require the user to input the desired target environments. Afterwards an appropriate wizard for the environment is opened to create the project and finally this project can be initialized to be used with SCCharts.

The advantage of this approach is that every installed project wizard can be extended to be used with SCCharts. For example, there is an Eclipse plugin for leJOS, which provides a project creation dialog to develop for the NXT brick. By installing this plugin and referencing its wizard, a concise project setup for using SCCharts together with leJOS can be created.

The required data for the introduced SCCharts wizard include

▻ a related project wizard,

▻ initial content for a file with wrapper code,

▻ initial content for wrapper code snippets.

Environments have already been identified as data containers for default settings. It makes sense to store information to initialize a project in environments as well.

4.4 Template Engines

In general a template engine is software that takes template files and mappings of variables as input and generates text files or streams as output [MVN06]. They are typically used to automatically generate code or web pages.

For example Java ServerPages (JSP) is a template system designed to dynamically generate the contents of web services, notably web sites [FKB01].

JSP provides a syntax to embed Java code in template files. Listing 4.1 shows simple code with JSP syntax and its generated output. The syntactical elements to start (`<%`) and end (`%>`) the embedded code are multiple characters wide. This is often used by template engines to avoid collisions with the target language, which is HTML in the example.

Like most template engines, JSP has its focus on web sites. However, the same mechanics can often be used to generate arbitrary text output, including wrapper code for SCCharts. To automatically generate wrapper code, users have to be able to write and modify templates on their own because the required snippets highly depend on the target device and programming language used.

Hence there are several requirements for a template engine to be used in the KIELER project:

- ▻ It has to be a general purpose engine, not limited to web pages.
- ▻ Its syntax must be intuitive, possibly independent of programming languages. It should not be necessary for users to learn a complete new programming language to write code snippets.
- ▻ There should not be any redundancy in wrapper code definitions, following the *don't repeat yourself* principle.
- ▻ There should be user defined parameters for snippet definitions. In contrast to a fixed set of arguments (e.g. arg1, arg2, arg3, etc.), user defined parameters in code snippets increase readability and adapt better with the requirements of the component the wrapper code is made for.
- ▻ Using annotations in a model file and its snippet declaration in a template file should have a similar syntax as this is more consistent.
- ▻ A Java API for the template engine is needed because KIELER is developed using Java.
- ▻ The engine should be open-source and come with a minimum of dependencies regarding third party products to retain the free, open-source status of KIELER.
- ▻ The engine has to be stable and approved in other projects.

Less critical is the engine's performance as wrapper code generation will be part of a project launch, which at the same time compiles model files concurrently via KiCo. As a result, wrapper code generation will not drastically increase the overall compilation time. Additionally, the desired wrapper code generation is text replacement, possibly with control structures such as conditionals. Thus it is a relatively easy task for an engine that has been optimized to generate HTML code for lots of web site requests as fast as possible.

```
<h1>Weather Forecast</h1>
<% for (int i=1; i<4; i++) { %>
   <p>Day <%= i %>: Fantastic</p>
<% } %>
```

(a) Template file with embedded Java code

```
<h1>Weather Forecast</h1>
   <p>Day 1: Fantastic</p>
   <p>Day 2: Fantastic</p>
   <p>Day 3: Fantastic</p>
```

(b) Output produced by JSP

Listing 4.1. The example shows a Java for-loop, embedded in an HTML snippet, and the resulting output. JSP evaluates the code between `<%` and `%>` as Java code. `<%= i %>` is used to evaluate the variable `i`.

Advanced techniques that some engines provide, such as template inheritance, are not necessary for the desired wrapper code generation because code snippets will not be partially overwritten in a single project.

There are plenty of template engines available.[1] In the following we will take a more detailed view on several template engines for Java. Thereby we will only evaluate open-source engines that are beyond its alpha and beta release. At the time of writing Histone[2] and Rythm[3] are in an alpha state and thus will not be reviewed any further.

4.4.1 Casper

Casper is a small template engine, consisting of a single Java Archive (JAR), with a syntax similar to JSP.[4] The engine has a simple Java API. However, it uses JavaScript (JS) in the background such that the evaluation of a template first results in JS code which is interpreted again afterwards. This enables the usage of JS code in template expressions but also introduces an overhead. Furthermore Casper is a small engine which does not seem to be maintained any longer. The last commit to its source code has been 5 years ago in 2010. The documentation of Casper is a short user guide which is only suited as introduction.

4.4.2 ChunkTemplates

ChunkTemplates is contained in a single JAR and suited to produce arbitrary text output.[5] A single file may contain several template definitions, which can be called programmatically as shown in Listing 4.2.

The documentation of ChunkTemplates explains its principles, syntax and usage in detail and there are lots of simple interactive examples. There is also an Eclipse plugin for

[1] `https://en.wikipedia.org/wiki/Comparison_of_web_template_engines`

[2] `http://weblab.megafon.ru/histone/en`

[3] `http://rythmengine.org`

[4] `https://code.google.com/p/casper`

[5] `http://www.x5software.com/chunk`

```
{#welcome}
Hello {$name}! Welcome!
{#bye}
Bye {$name}! We will miss you!
```

(a) ChunkTemplates file: dir/hello.chtml

```
Hello Bob! Welcome!
```

(c) ChunkTemplates example result

```java
Theme theme = new Theme("dir");

Chunk welcomeTemplate = theme.makeChunk("hello#welcome");
welcomeTemplate.set("name", "Bob");
welcomeTemplate.render(System.out);
```

(b) Java API for ChunkTemplates

Listing 4.2. The example defines two templates, `welcome` and `bye` in the file `snippets/hello.chtml`. The first template is evaluated using the Java API.

ChunkTemplates that provides syntax highlighting and an outline of templates.

However, the syntax to evaluate a variable, which is `{$name}`, differs from using a variable inside Eclipse, which is `${name}`.

By design, ChunkTemplates does not throw errors if variables are undefined. The placeholder will just be carried to the output. Alternative one can provide a default value or an error message explicitly. Although this might be suited for a web site, it is not a reasonable behavior when generating wrapper code. On the one hand explicit null-checking is a tedious task. On the other hand, without an explicit null-check an error will be thrown at a later time as the output is not a correct program. Thus the time required to find such bugs is increased.

Furthermore, ChunkTemplates does not provide a syntax to define functions or macros with a set of parameters. A macro in ChunkTemplates is a way to call template definition A from template B. However, this requires the knowledge of the variable names in template A. Parameters of function calls are determined by their position in most programming languages. However, macro calls in ChunkTemplates are bloated with parameter names as seen in Listing 4.3b.

Although users of wrapper code generation in KIELER should be able to define their own placeholders in template files, this will require the knowledge of the placeholder names on the calling Java site. However, this makes the implementation more complex because required parameter names have to be parsed manually.

```
{#LightSensor}
static LightSensor sensor${port} =
   new LightSensor(SensorPort.${port});
```

(a) ChunkTemplate example: sensors.chtml

```
{% exec sensors#LightSensor %}
   {$port = S3}
{% endexec %}
```

(b) Macro call in ChunkTemplates

Listing 4.3. The first listing defines a template for wrapper code of a light sensor in leJOS. In the second listing the template is called using the exec command and the variable `port` is set using its name.

4.4.3 Pebble

Pebble is a template engine created by Mitchell Bösecke.[6] It is actively developed and its documentation is detailed and has a modern presentation.

Pebble is not distributed as JAR file but via Maven Central Repository.

Similar to ChunkTemplates, evaluating a variable in Pebble differs from using variables in Eclipse, which can be seen in Listing 4.4.

It is possible to define parameterized macros in Pebble. The parameters may have default values. Afterwards macros can be used similar to functions, either without explicit parameter names or with parameter names so that their order becomes irrelevant.

However, there are some downsides to the way Pebble handles macros. Firstly, macro definitions and calling a macro uses parenthesis to surround the parameters. This is a common syntax for functions in programming languages. However, it differs from the annotation syntax in SCCharts, where parameters for an annotation are separated via spaces and commas.

Secondly, macros can only access their local variables. This makes it difficult to provide meta information about the variable a snippet is used for in the context of wrapper code generation (e.g., the variable's name and type). These information would have to be a parameter in every macro.

4.4.4 StringTemplate

StringTemplate is a strict template engine with a formal definition of its grammar and purpose.[7] It is a strict template engine in the sense that it does not provide features known from imperative programming languages such as assignments, loops, arithmetic expressions and arbitrary method calls.

This has been done by design as these features invite developers to put logic in templates which violates the separation of model and view. The features StringTemplate offers are variable evaluation, template inclusion, macro definitions, conditionals (if statements)

[6] `http://www.mitchellbosecke.com/pebble/home`

[7] `http://www.stringtemplate.org`

```
{# THIS IS A COMMENT #}
{% macro printError(message="default error message") %}
   <p>An error occured: {{ message }}</p>
{% endmacro %}

{% if category == "news" %}
   {{ news }}
{% else if category == "sports" %}
   {{ sports }}
{% endif %}
```

(a) Pebble example: template.peb

```java
PebbleEngine engine = new PebbleEngine();
PebbleTemplate compiledTemplate = engine.getTemplate("template.peb");

Writer writer = new StringWriter();

Map<String, Object> context = new HashMap<>();
context.put("category", "news");
context.put("news", "The weather is nice.");
compiledTemplate.evaluate(writer, context);

String output = writer.toString();
```

(b) Java API for Pebble

```
The weather is nice.
```

(c) Pebble example result

Listing 4.4. The example shows the usage of a macro as well as the primary tags `{# #}`, `{% %}` and `{{ }}`. The first tag marks a comment, whereas the second is used to change the control flow in a template. The third tag evaluates an expression that shall appear in the output.

and iterating over lists. These features do not make a Turing-complete language, but can already produce the class of context-free languages.[8]

[8] https://theantlrguy.atlassian.net/wiki/display/ST4/Motivation+and+philosophy

StringTemplate is contained in a single JAR but does require ANTLR[9], which is used to parse the template files. The documentation provides small examples and includes formal grammar definitions.

StringTemplate tries to reuse syntax introduced in other languages and formalism. However, it still requires a second look to understand templates in the beginning. This is because the character used to mark an expression is not fix. They can be set in a file using the `delimiters` keyword, which is illustrated in Listing 4.5.

Although StringTemplate is a well thought out template system and its emphasis of model-view separation may be advantageous in the development of large web sites and systems, its restriction not to have variable assignments in templates make it a more challenging tool for wrapper code generation in KIELER.

4.4.5 Apache Velocity

Apache Velocity is another free, open-source template engine for Java.[10]

It consists of two JAR files and enables the usage of Java methods in templates. The JAR files contain other libraries which Velocity depend on. These include Jakarta Commons Collections and Jakarta Commons Lang, which provide additional API for Java collections and core classes such as Strings.

There is syntax to evaluate variables and for directives, which are used to change the control flow or assign values to variables. To evaluate a variable it has to be prefixed with a dollar sign. Directives start with a hash.

The Velocity documentation features small explanations and examples on how to use the syntactical elements.[11] Additionally there are examples how Velocity can be used for the generation of websites.

It is possible to define macros in Velocity. These can have parameters and may be called with additional content as shown in Listing 4.6

However, it is not possible to provide default values for macro parameters. Furthermore, as already seen in ChunkTemplates, undefined references to variables do not throw an error per default. Instead the placeholder is just taken to the generated output.

The last stable release of Velocity (version 1.7) is from 2010 for which an Eclipse plugin is available.

[9] http://www.antlr.org

[10] http://velocity.apache.org/engine/index.html

[11] http://velocity.apache.org/engine/releases/velocity-1.7/user-guide.html

```
delimiters "<", ">"
// single-line template
brackets(x) ::= "(<x>)"
// multi line template
t2(a, b) ::= <<
   <if((!a||b))>yes<else>no<endif>
>>
// multi line template that ignores indentation and newlines
sayHi(name) ::= <%
   Hi <name>!
%>
```

(a) StringTemplate example: templates.stg

```
STGroup group = new STGroupFile("templates.stg");
ST hello = group.getInstanceOf("sayHi");
hello.add("name", "Elsa")
System.out.println(hello.render());
```

(b) Java API for StringTemplate

```
Hi Elsa!
```

(c) StringTemplate example result

Listing 4.5. The first listing shows a file with several template definitions. At the very beginning the characters to frame an expression that should be evaluated is set to angle-brackets.

4.4.6 FreeMarker

FreeMarker is a template engine which has been used in several applications such as the Netbeans IDE.[12] The engine has a detailed documentation. It contains examples that are explained step by step to introduce features. Additionally, concepts of template engines are explained in general and afterwards how they are implemented in FreeMarker. The documentation is up to date and its presentation uses colored backgrounds as well as syntax highlighting to visualize listings and additional information.

Semantically FreeMarker offers features similar to those seen in other engines. These include basic tasks such as evaluation of variables, including other templates, macro

[12] http://freemarker.org/poweredBy.html

```
#macro( paragraph $title )
   <h2>$title</h2>
   <p>$bodyContent</p>
#end

#@paragraph("Weather News")The weather is great!#end
```

(a) Apache Velocity example: weather.vm

```
VelocityEngine ve = new VelocityEngine();
ve.init();
Template t = ve.getTemplate("weather.vm");
StringWriter writer = new StringWriter();
template.merge( context, writer );
```

(b) Java API for Velocity

```
<h2>Weather News</h2>
<p>The weather is great!</p>
```

(c) Velocity example result

Listing 4.6. The template file defines a macro and calls it with an argument and an additional body.

definitions, conditionals and iterating over collections. Further, it is possible to set variable values in templates and do arithmetic and boolean calculations.

Syntactically it uses a consistent XML-based syntax. This makes it a rather easy to understand language as the basics of XML are known by the majority of programmers. To evaluate a variable the syntax is `${variable_name}`, which is the same as in the Eclipse IDE.

As in Velocity, macros can be called with an additional body. However, in FreeMarker this seems much more natural because of its XML-based syntax. Listing 4.7 illustrates this. Furthermore, macros can have default parameters and since the parameter list is not surrounded by braces it is similar to annotations in SCCharts.

There is also some syntactic sugar that deviates from XML. Macro calls can be ended with `</@>` independent of the macro's name. If a FreeMarker tag does not have an inner body and would always be written with an ending slash, the slash is optional. Thus the tag `<#nested />` can be written as `<#nested>`.

Errors are thrown with information where and why the error occurred and often with

```
<#macro paragraph title="No Title">
   <h2>$title</h2>
   <p><#nested></p>
</#macro>

<@paragraph "Weather News">The weather is great!</@paragraph>
```

(a) FreeMarker example: weather.ftl

```
Configuration cfg = new Configuration(Configuration.VERSION_2_3_22);
cfg.setDirectoryForTemplateLoading(new
    File("/where/you/store/templates"));
cfg.setDefaultEncoding("UTF-8");

Map data = new HashMap();
Template template = cfg.getTemplate("weather.ftl");
StringWriter writer = new StringWriter()
template.process(data, writer);
```

(b) Java API for FreeMarker

```
<h2>Weather News</h2>
<p>The weather is great!</p>
```

(c) FreeMarker example result

Listing 4.7. The template file defines a macro and calls it with arguments and an additional body.

a hint, how it can be fixed. Per default, undefined variables and expressions raise errors which makes it easier to find bugs.

FreeMarker is actively developed and contained in a single JAR file without further dependencies. There is an Eclipse plugin as part of the JBoss Tools[13] which provides an editor with syntax highlighting and static error analyzes.

[13] http://tools.jboss.org/

4.4.7 Xtend and Xtext

In general it would be easier to use tools which are already known and proven in the development of KIELER instead of using an additional third-party engine.

Xtend has a feature called *template expressions*. These can be used to embed code in a string and provide a well-thought-out whitespace handling for indentation. However, it is only a tool for advanced string concatenation within Xtend, but not a full template engine. Thus it is not possible to load and evaluate templates from the file system with user defined macros. For the desired wrapper code generation these features are essential because users have to be able to extend and modify wrapper code snippets by themselves.

Another option is to write a template engine specific for wrapper code generation in KIELER. This can be done using the Xtext framework, which has been used to define the grammar of SCT files. Anyhow, this is not necessary. There are already several proven template engines as seen above.

4.4.8 Conclusion

Requirements for a template system have been named so that it can be used in KIELER to generate wrapper code from annotated model files. Several open-source engines are available for Java. Listing 4.1 gives a summary of the key points of the introduced template engines.

FreeMarker does meet all requirements. Its XML-based syntax is familiar to most programmers. This is a benefit as users should be able to modify templates by themselves without the need to learn a complete new language. FreeMarker is also still maintained and a proven, feature rich engine. Furthermore, its documentation is detailed and well structured and its Eclipse plugin eases working with templates.

In conclusion, FreeMarker will be used as template engine to generate wrapper code from annotated SCCharts.

4.5 Wrapper Code Generation

SCT files may contain annotations that are not intended to be used for wrapper code generation. For instance, there are annotations to manipulate how the SCChart is rendered in KIELER. As a result, annotations may be used that are undefined in the context of wrapper code generation. One solution is to use annotations implicitly and ignore it if the corresponding code snippet is not defined. However, this would bypass a strong error response. For example a typographical mistake in an annotation would not raise an error anymore. Another solution is to use an explicit wrapper code annotation. When using annotations for wrapper code exclusively, this explicit annotation may be an inconvenient repetition. As compromise both solutions will be implemented. If the explicit annotation

Table 4.1. Comparision of open-source template engines for Java

Engine	Pro	Contra
Casper	▻ Single JAR file	▻ JavaScript as intermediate step ▻ Little documentation ▻ Not maintained
Chunk-Templates	▻ Eclipse plugin ▻ Single JAR file	▻ No error on undefined variables ▻ Set macro argument only by name
Pebble		▻ Installation only via Maven ▻ Macros can only access local variables
String-Template	▻ Eclipse plugin	▻ Requires ANTLR ▻ Not possible to set variable value in template
Velocity	▻ Eclipse plugin	▻ Depends on other libraries ▻ No release since 2010
FreeMarker	▻ Eclipse plugin ▻ Single JAR archive ▻ XML-based syntax ▻ Excellent documentation ▻ Actively developed ▻ Undefined variables throw errors per default	

`@Wrapper` is used, errors will be thrown if the following snippet name is undefined. Thus, the user has the freedom to decide which approach is appropriate for his project.

As wrapper code annotations can have parameters, it must be possible to define parameterized snippets as well. The macro feature of FreeMarker is suited for this.

A single wrapper code annotation may require multiple code snippets, depending on its purpose. To illustrate, the annotation `@MotorSpeed` may set the speed of a motor if it is used on an output variable. If it is used on an input however, it makes sense to read the current motor speed and set the input variable accordingly. Furthermore, there might be a one time initialization necessary, e.g., to create a motor object before it can be used.

In conclusion, a single wrapper code snippet may consist of three parts:

1. code for initialization,
2. code for an input variable,
3. code for an output variable.

To define the different parts of a snippet one can utilize conditionals in a FreeMarker macro. However, repeating the same if-statement in all snippets is not a desirable solution. A workaround can be implemented using FreeMarker macros itself as seen in Listing 4.8. Using this approach, it is possible to define the different parts of wrapper code snippets in a declarative manner. It turns the snippet definition from *"if the initialization part is needed then use this code"* to *"this is the initialization part"*.

Macros defined this way only need to be called in a template file in an appropriate place when the corresponding annotation is found in a model file. Therefore the template file will be sent twice to the FreeMarker engine. The first time placeholders for the macro calls are replaced. Afterwards this new template is sent to FreeMarker to evaluate the macros which results in the desired wrapper code.

This process is illustrated by an example in Listing 4.9. Firstly, there is an annotated model file, which is shown in Listing 4.9a. Secondly, there is the static template file with placeholders for macro calls, which is shown in Listing 4.9b. These are the input files for wrapper code generation. As part of the project launch, annotations from the model are transformed to FreeMarker macro calls. These macro calls are sent to FreeMarker, which substitutes the appropriate placeholders in a first iteration. The result is shown in Listing 4.9c. In the second iteration, FreeMarker evaluates the macro calls so that the result is the desired wrapper code as seen in Listing 4.9d.

```
<#macro init>
   <#if phase=='init'>
       <#nested>
   </#if>
</#macro>

<#macro input>
   <#if phase=='input'>
       <#nested>
   </#if>
</#macro>

<#macro output>
    <#if phase=='output'>
        <#nested>
    </#if>
</#macro>

<!-- Snippet definition example -->
<#macro Floodlight port>
    <@init>
        LightSensor lightSensorFloodlight${port} =
            new LightSensor(SensorPort.${port});
    </@>
    <@input>
        scchart.${varname} = lightSensorFloodlight${port}.getFloodlight();
    </@>
    <@output>
        lightSensorFloodlight${port}.setFloodlight(scchart.${varname});
    </@>
</#macro>
```

Listing 4.8. FreeMarker macros to define the different parts of a wrapper code snippet and their usage.

```
...
@Floodlight S3
output bool lightOn;
...
```

(a) Excerpt of an annotated SCT

```
...
// Initialization
${inits}
// Tick loop
while(true){
   // Input snippets
   ${inputs}
   // Reaction of model
   scchart.tick();
   // Output snippets
   ${outputs}
}
...
```

(b) Excerpt of a template file

```
...
// Initialization
<assign phase = 'init'>
<assign varname = 'lightOn'
   vartype = 'bool'>
<@Floodlight S3>
// Tick loop
while(true){
   // Input snippets

   // Reaction of model
   scchart.tick();

   // Output snippets
   <assign phase = 'output'>
   <assign varname = 'lightOn'
      vartype = 'bool'>
   <@Floodlight S3>
}
...
```

(c) First iteration on the template file

```
...
// Initialization
LightSensor lightSensorFloodlightS3 =
   new LightSensor(SensorPort.S3);
// Tick loop
while(true){
   // Input snippets

   // Reaction of model
   scchart.tick();

   // Output snippets
   lightSensorFloodlightS3.
      setFloodlight(scchart.S3);
}
...
```

(d) Second iteration on the template file

Listing 4.9. Step by step wrapper code generation. The first iteration inserts macro calls in the template which are evaluated by FreeMarker in the second iteration, yielding the desired wrapper code.

Chapter 5

Implementation

This chapter explains the implemented plugins in detail. The new plugins provide features for an improved workflow and project management and are thus named *Prom*. Overall three plugins have been developed in this thesis:

- ▷ **org.freemarker** A wrapper for the FreeMarker engine that provides its API to use in other plugins.
- ▷ **de.cau.cs.kieler.prom** A plugin with general Prom features and Extension Points to use these for specific modeling languages.
- ▷ **de.cau.cs.kieler.sccharts.prom** A plugin that utilizes the features of Prom to implement project management specifically for SCCharts.

5.1 FreeMarker Plugin

The FreeMarker plugin contains a single class, which handles the plugin's life-cycle within the Eclipse framework. Additionally it contains API to create a new FreeMarker configuration object directly for a folder with template files. The class also contains macro definitions of `<@init>`, `<@input>`, and `<@output>` such that they can be used within wrapper code snippets to declare their different parts as seen in Figure 4.8.

Besides the aforesaid class, the plugin contains the FreeMarker library and makes it visible to other plugins that reference it.

5.2 General Project Management Plugin

The general Prom plugin contains utility methods for the UI, data containers, the new launch configuration, environments and their UI, as well as the basis for easy addition of file wizards and project wizards. Further, Prom adds Extension Points for initialization of new environments and to analyze the annotations of model files in the context of wrapper code generation.

The dependencies on other plugins are visualized in Figure 5.1 whereas reasons for the dependencies are explained in Table 5.1.

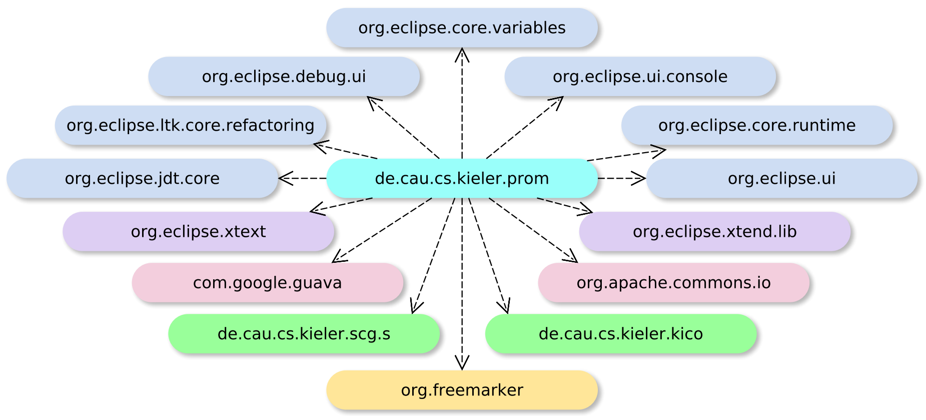


Figure 5.1. The dependencies of the general prom plugin. Blue is for Eclipse plugins, purple are libraries to use Xtend, pink are utility plugins that provide useful API in general, green are dependencies to other KIELER plugins, and orange is the new plugin for FreeMarker.

The plugin broadly separates in five packages: `launchconfig`, `filewizard`, `projectwizard`, `environments` and `common`. This is illustrated in Figure 5.2. The `common` package contains utility and data container classes such that it is referenced by the others. The `environments` package is used to initialize launch configurations as well as projects.

Table 5.1. Prom plugin dependency explanations

Plugin	**Reason for dependency**
org.eclipse.core.runtime	Core functionality for Eclipse plugins
org.eclipse.ui	Required plugin to provide UI additions in Eclipse
org.eclipse.debug.ui	Plugin containing Extension Points for launch configurations.
org.eclipse.ui.console	Provides API to communicate with a launched process in the Console View
org.eclipse.core.variables	Contains features used to utilize placeholders for the main file in the new launch config
org.eclipse.jdt.core	Used to set the related project wizard for a default *Generic Java* environment
org.eclipse.ltk .core.refactoring	When renaming and deleting files in a project, the corresponding launch config for KiCo adapts automatically using Extension Points from this plugin

org.eclipse.xtext org.eclipse.xtend.lib	Required for developing in the Xtend language
de.cau.cs.kieler.kico	Used to compile SCT files with the KIELER compiler
de.cau.cs.kieler.scg.s	Required to fetch the supported code targets from KiCo
com.google.guava org.apache.commons.io	Provides useful API for Java development in general
org.freemarker	The new FreeMarker plugin used for wrapper code generation

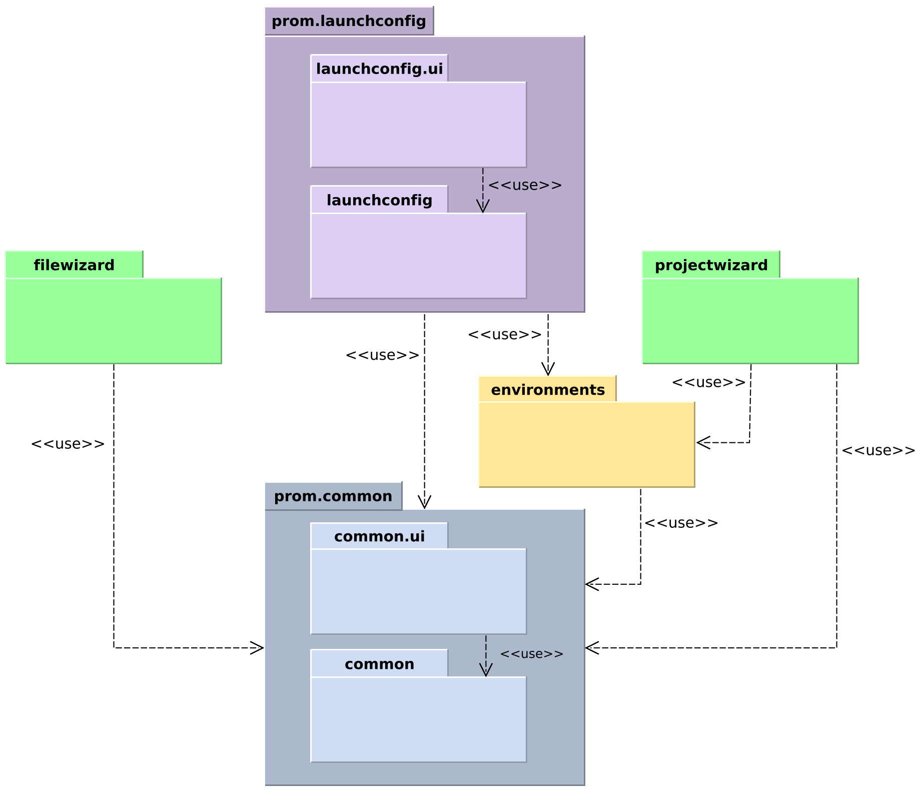


Figure 5.2. Overview of the packages within the Prom plugin.

5.2.1 The Common Package

`Common` contains utility classes and data containers as well as the class which handles the life-cycle for the plugin. All data containers extend `ConfigurationSerializableData`. This abstract class implements methods to persist its fields in a launch configuration or preference store. Therefore it uses Java reflection to iterate over its fields of type string. Eclipse provides API to persist strings. However, this API differs for launch configs and preferences such that the implemented class is used as abstraction for both. Non string fields, such as lists, need to be saved by extending classes manually.

To uniquely identify the fields of a persistable data object, the class contains a getter for an identifier that subclasses may override. For instance, the identifier for an environment is its name whereas for a model file it is its path. Thus the getter will return those fields. Identifiers may not contain a comma because this character is used to separate them when using a single string.

There are several data containers which extend the base class.

- **CommandData** stores the attributes of commands from the list of user defined shell commands in a launch config.
- **EnvironmentData** stores the fields of environments, including a list of CommandData.
- **FileCompilationData** stores data of model files that should be compiled.

Utility classes that are implemented in `common` are

- **ExtensionLookupUtil** which is used to search for Extensions—such as project wizards—in the Eclipse plugin registry.
- **ModelImporter** which takes care of loading model files, e.g., SCT.

The `common.ui` package provides the class `UIUtil` which eases the creation of the GUI. This ranges from creating simple composite containers, over a combobox with the KiCo targets, to more complex GUI compositions such as a text field with a browse button next to it, which is used to modify the text field.

Besides the creation of the UI, there are also methods to enable and disable controls recursively. All methods in `UIUtil` are implemented as static.

5.2.2 Environments

There are several classes that make up the implementation of Environments. Firstly, there is the data container `EnvironmentData`. Secondly, there is the UI to create and modify environments which is implemented in `EnvironmentsPage`. This class extends a base class for preference pages and is referenced in a corresponding Eclipse Extension. The page in the preferences consists of a list of environments with buttons to add, remove and change the order of these. Further, there is a group of tabs which categorize the corresponding controls for the fields of environments. They orient on the tabs for the launch configuration.

The third entity that make up Environments is the `IEnvironmentsInitializer` interface. This is used in an Extension Point so that other plugins may provide a list of default Environments. The initialization is done when no Environment is stored in the preferences. This is the case on a newly installed Eclipse. Finally there is `PromEnvironmentsInitializer` which implements aforesaid interface to provide generic Environments for KiCo compilation of models to the C and Java language.

5.2.3 Project Creation Dialog

The project wizard implemented in Prom is not referenced directly in an Extension. Instead it provides the basis for project wizards for a specific modeling language, of which the KIELER Compiler can generate code from. Therefore an extending project wizard needs to define the file extension and default contents for its model file. This is done by setting `modelFileInitialContentURL` such that it points to a valid file path or URL with the desired default content.

Eclipse provides an URL protocol such that files from a plugin can be directly addressed. For instance, to reference the file *resources/default.sct* in a plugin *de.cau.cs.kieler.sccharts.prom* the URL has the form *platform:/plugin/de.cau.cs.kieler.sccharts.prom/resources/default.sct*. Although this works for files within an Eclipse plugin, it is not possible to directly load the contents of a folder this way. Thus, if a directory reference in a launch config uses the platform protocol then the contents of this directory are loaded via an own implementation, which iterates over the folder's files.

Environments reference the related project wizard via the fully qualified name of its implementing class. This entry is used by the Prom project wizard to delegate the instantiation of this class to `ExtensionLookupUtil` from the `common` package.

However, performing this task when the user clicks *Finish* on the wizard can be considered bad user guidance as it opens further dialogs and does not finish the wizard as one might expect. Instead, opening the related project wizard is done when the user clicks the *Next* button. To do so, a dummy page without content is added to the Prom project wizard. Otherwise the *Next* button would be disabled.

Should the user cancel the wizard at any point, the newly created project is deleted. Thus errors do not abort the wizard, but prevent its closure so that the user needs to cancel the wizard and thereby removes unfinished initializations. This gives the user the freedom to alter settings of the wizard even after an error occurred.

In conclusion, to add a project wizard for a new modeling language, one simply needs to extend the base class provided by Prom, set `modelFileExtension` and `modelFileInitialContentURL` and finally reference this implementation in a `newWizard`

Extension. All other features—creating a project by using a related wizard as well as initializing the project with a model file, main file and wrapper code snippets— are done by the base implementation.

5.2.4 Launch Configuration

The launch configuration is separated in the plugins `launchconfig` and `launchconfig.ui`. The UI plugin contains the tab group implementation which Eclipse uses to communicate launch configs to the user. Each tab is implemented in its own class and data containers from the `common` package are used to persist the launch configuration settings.

When a launch config is started, the class `LaunchConfiguration` is instantiated and its method for launching is called. This is the heart of the launch configuration. At first it creates a console for the Console View, where all errors are reported. Afterwards it loads all settings and checks that the project exists.

If the project exists then the placeholder variables for the main file are set which can be used in shell commands of the launch config. The main file and *compiled main file* have four placeholders each. These are the name, name without file extension, project relative path and absolute file system path. The compiled main file will be saved in the *kieler-gen* directory in contrast to the non compiled main file.

After the variables are set, two jobs are created. Creating a job is the standard way to implement concurrency in Eclipse. Jobs can run in their own thread and have several UI features in Eclipse, e.g., for progress indication and abort. The first job in the launch config is for the compilation of model files in KiCo. The second job takes care of wrapper code generation from models. Both are started at the same time and thus may be executed concurrently. After both finish successfully, the list of shell commands for the launch configuration is executed sequentially. The described process corresponds to Figure 4.2.

To keep the `LaunchConfiguration` class at a maintainable size and for separation of concerns, the wrapper code generation and execution of shell commands are externalized in the classes `WrapperCodeGenerator` and `CommandExecutor`. They are instantiated in the corresponding jobs.

The generic Prom plugin is meant to be independent of the model language used. Thus the `WrapperCodeGenerator` does not know which are annotations for wrapper code generation in a model file. Instead, reading this information needs to be implemented by other plugins. Therefore Prom provides an Extension Point `wrapperCodeAnnotationAnalyzer`. and an interface `IWrapperCodeAnnotationAnalyzer`

Further, there is a launch shortcut in the package. It can be used on any model file. However, providing the Extension which adds this shortcut, e.g., for an SCT file, needs to be done in other plugins. When the launch shortcut is run on a model file X it will first search for a KiCo launch configuration used for the project that X is part of. If such a config is found then X is added to its list of model files and the configuration is launched.

However, if no such config could be found the shortcut will create a new one for the project and add *X* to its list of model files. To initialize the new configuration, the environment used to set up the project of *X* is fetched from the project properties. If this information can't be found, a dialog will open and query an appropriate environment from the user. Afterwards the launch configuration is initialized with the data from this environment. In analogous manner the shortcut will look for or query the project's main file and add this information to the configuration.

There are two remaining classes in the `launchconfig` package, of which the first is `LaunchConfigRenameParticipant`. The class is used in an Extension that triggers when files or projects are renamed. It will then iterate over all KiCo launch configurations and adapt file paths for the main file and model files to the new name. Thus, renaming a model file or main file does not break a launch config.

Analogously `LaunchConfigDeleteParticipant` removes paths or deletes a launch configuration entirely if the corresponding file or project is deleted.

5.3 SCCharts Project Management Plugin

The Prom plugin specifically for SCCharts has new dependencies on other plugins, namely `de.cau.cs.kieler.sccharts.text` and `de.cau.cs.kieler.prom`. The dependency on Prom is obvious. The other plugin contains the SCCharts language classes which are used in an annotation analyzer to identify annotations for wrapper code generation. Other plugin dependencies are already known from the generic Prom plugin.

In contrast to the generic Prom plugin, there is only one package containing the classes `SCChartsPromPlugin`, `SCChartsEnvironmentInitializer`, `SCChartsFileWizard`, `SCChartsProjectWizard` and `SCTWrapperCodeAnnotationAnalyzer`.

`SCChartsPromPlugin` handles the plugin's life-cycle. The other classes are implementations of Extensions, which use interfaces or base classes defined in the generic Prom plugin.

The Prom plugin for SCCharts is rather simple and contains only small classes compared to the generic Prom plugin. This speaks for a good separation of both plugins and shows that adding project management features for a new modeling language is a relatively easy task.

Chapter 6

Evaluation

In the following we will discuss advantages and disadvantages of the solution. Further, we will see results of a small survey, which has been conducted to evaluate the new workflow.

6.1 Advantages and Disadvantages

At first Prom introduces an overhead because an environment has to be set up. However, this has to be done only once and generic environments for Java and C are included in the Prom plugin.

A dangerous disadvantage is that bugs in wrapper code templates might spread. This occurs when code snippet definitions with bugs are copied to new projects. To fix these, all projects using the snippets would have to be revised. To prevent such a situation, it is possible to keep snippets for an environment in a single, project independent directory. This directory can then be referenced in launch configurations making bug fixes necessary only in a single place. Further, it is easily possible to keep such a directory under version control. This is especially desirable when working in a team.

Another disadvantage regarding wrapper code generation is that users need to know the defined snippet macros and their parameters. Otherwise they will not be able to utilize them. On the other hand, without snippet definitions knowledge of the API for sensors and actuators is required to write wrapper code manually. Thus this is a minor disadvantage.

A positive aspect of the new Prom plugin is that a runnable project structure can be directly created. In contrast to the old workflow, there is no need to manually create and initialize an SCT file and main file. This behavior can be achieved with every installed project wizard because they are referenced in creating an SCCharts project.

Furthermore, writing wrapper code becomes optional, if all required code snippets are present. Anyhow, snippet definitions can be easily extended and modified. Although NXC and leJOS provide a fundamentally different API for sensors and actuators, it is possible to write wrapper code snippets that perform equivalent tasks and require the same annotations in an SCT file. Thus, by using wrapper code generation, the same model can be used for different environments if all required wrapper code snippets are present.

Finally, the new launch configuration provides short code-test iterations. The list of arbitrary commands is a suited solution for many use-cases to further compile and

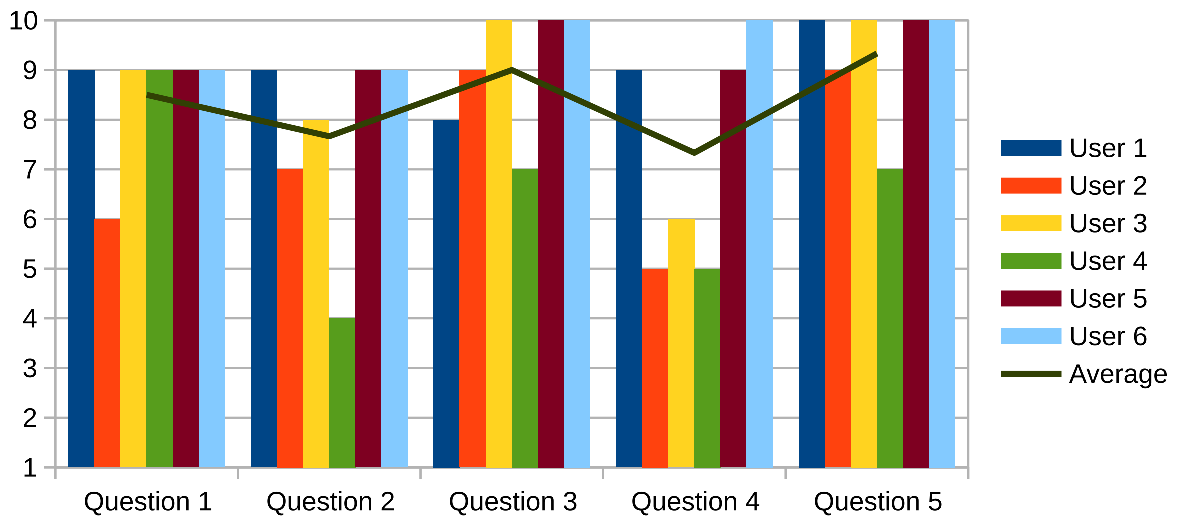


Figure 6.1. Result of the survey. 6 Users have been tested. The questions have been asked such that the higher a ranking is, the better is the implemented solution.

deploy results from KiCo compilation. Additionally it is possible to only use the launch configuration to compile SCT files with KiCo. Afterwards complex compilation tasks can be composed by using this KiCo launch configuration together with launch groups provided by the CDT.

6.2 Survey

To compare the old and new workflow, small user experiments have been conducted.

The participants were tested independently, starting with a short introduction to the new Prom features. Annotations in SCT files, wrapper code snippets and launching a project were explained by using an example project. Afterwards each test person was assigned a task that required project creation, modeling an SCChart, using the correct wrapper code annotations and finally launching the created project. The survey task and questions are listed in Appendix A. Not part of the experiment was the creation of a new environment.

The test ends with a survey consisting of five questions that were answered with a rating from 1 to 10. All questions have been asked such that the higher the rating, the better is the new Prom plugin. The questions are:

1. How intuitive did you, as an experienced Eclipse user, find the project creation and launch? (1 means *Prom is overly complicated*, 10 means *the setup does not require a second thought*)

2. How well does the overall solution blend with other Eclipse tools? (1 means *Prom UI elements are directly identifiable*, 10 means *no difference to other plugins can be seen*)
3. Would you recommend wrapper code generation from annotations or hand written code? (1 means *always hand written code*, 10 means *always code generation*)
4. How easy do you think it is to modify the wrapper code templates? (1 means *very difficult*, 10 means *very easy*)
5. If you have used the former approach, how does it compare with the new project management? (1 means *the old approach is much better*, 10 means *the new management is much better*)

Altogether each experiment took about 15 to 20 minutes and 6 people where tested.

The results are presented in Figure 6.1. No test person perceived the old manual workflow as superior and nearly all would recommend using wrapper code from annotations. Further, extending the snippet definitions has been perceived as at least feasible by the test persons. This indicates that the XML-based syntax for wrapper code snippets is intuitive.

The visual representation of the new functionality has been evaluated with the second question. Although the response is positive in general, there are testers that have rated it significantly less high. This might result from a different level of experience using Eclipse.

Anyhow, the resulting projects did not require any manual creation of wrapper code, although the final main file had a length between 29 to 50 lines of code without comments. The created model files contained about 30 lines of code so that about half of the overall required code was created by Prom. Thus it can be considered a noteworthy simplification in this case. In general this factor depends on the number of required input and output annotations and the complexity of the model. The smaller the model file and the more input-output mapping has to be performed, the greater is the benefit of automatic wrapper code generation via Prom.

In conclusion the implemented features can be considered an improvement to the former workflow, notably for small models.

Chapter 7

Conclusion

This chapter summarizes the thesis and afterwards discusses possible additions to the introduced concepts as future work.

7.1 Summary

In Chapter 1 we have seen that KIELER did not provide a streamlined workflow when developing with SCCharts. Project setup, compilation and deployment involved user actions, that could be automated. Further, manually writing wrapper code distracted from the modeling itself.

The template engine FreeMarker has been identified as suited for wrapper code generation in KIELER. It stands out of other open-source engines with its feature rich, XML-based syntax, excellent documentation, and Eclipse support. Additionally FreeMarker is contained in a single Java archive and actively developed at the time of writing.

To improve the workflow in KIELER, a generic launch configuration has been designed. It compiles model files supported by KiCo, assembles wrapper code for these models and afterwards executes an arbitrary list of shell commands. Thus the new launch configuration is able to compile and deploy a project with a single user action. Environments are used to define default settings when developing for a specific target device and operating system. They can be configured in the preferences of Eclipse. A project wizard for SCCharts uses these environments to create and initialize projects.

The solution design has been implemented in three plugins for the KIELER tools. Firstly, there is a plugin that provides the functionality of the FreeMarker engine. Secondly, an abstract project management plugin provides the new launch configuration and the basis for file wizards and project wizards. Another plugin, specifically for SCCharts, builds upon this abstract implementation. It adds an SCT file dialog and a project wizard to the Eclipse platform. A launch shortcut was implemented to compile SCT files without setting up a launch configuration manually. Therefore, the shortcut utilizes default settings from environments. Furthermore, the plugin contains a class that analyzes annotations dedicated to wrapper code generation in an SCT file as part of a project launch.

For demonstration of the new tools, the Mindstorms NXT brick running with leJOS as well as NXC has been used. It shows that the same model file can be used for these different environments if all required wrapper code snippets are present, although the provided API of the two operating systems is fundamentally different. Small user studies verify that the new tools can noteworthy reduce the manual writing of wrapper code. Without the implemented plugins, about twice as much manually written code would have been required and all test persons perceived the new project management as superior compared to the former workflow. In conclusion, the implementation improves the workflow when using KIELER to model embedded systems with SCCharts.

7.2 Future Work

This thesis provides an improvement when working with SCCharts in KIELER. However, initializing a project is only possible with a model file, a main file and files containing wrapper code snippets. In future work this could be extended such that a project can be initialized with an arbitrary list of files and directories.

Currently wrapper code snippet definitions may consist of up to 3 parts. These are for initialization, to use snippets on an input variable, and to use snippets on an output variable. More flexibility can be achieved if the available parts can be configured directly by the user. For instance, it might be necessary to have an additional part to declare variables before they can be initialized.

Additional room for improvement can be found when moving files in a project that are referenced in a launch configuration for KiCo compilation. Moving such a file will break the configuration as the old file path will become invalid. However, Eclipse provides the Extension Point *moveParticipant*, which can be used to implement additional behavior when moving files. This can be used to implement an automatic adaption of file paths in KiCo launch configurations.

Appendix A

Survey Papers

The following pages contain the survey questions and task used to evaluate the implementation in a study.

Survey: SCCharts Modeling with Prom

1. How intuitive did you, as an experienced Eclipse user, find the project creation and launch?

1 = overly complicated 10 = no-brainer

1	2	3	4	5	6	7	8	9	10

2. How well does the overall solution blend with other Eclipse tools?

1 = everything identifiable 10 = can't see a difference

1	2	3	4	5	6	7	8	9	10

3. Would you recommend wrapper code generation from annotations or hand written code?

1 = hand written code 10 = code generation

1	2	3	4	5	6	7	8	9	10

4. How easy do you think it is to modify the wrapper code templates?

1 = very difficult 10 = very easy

1	2	3	4	5	6	7	8	9	10

5. If you have used the former approach, how does it compare with the new project management?

1 = old approach is much better 10 = new management is much better

1	2	3	4	5	6	7	8	9	10

Tasks: SCCharts Modeling with Prom

Task 1 – Dome Light

Create a new SCCharts project for Mindstorms NXJ.
In this project, create a program with SCCharts that will

1. turn on the floodlight if at least one touch sensor is pressed
2. if no touch sensor is pressed anymore: let the floodlight stay on for 3 seconds and afterwards turn it off
3. repeat

Launch the project.

Appendix B

User Manual

The following pages contain the documentation of the developed plugins for end users.

B.1 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 initialize and run the model. To do so, there are mainly three components: An Eclipse *launch configuration*, so called *environments*, and *project wizards*, which will be introduced in the following.

B.2 The KiCo Launch Configuration

Prom provides a launch configuration (launch config) to

1. compile code from models via KiCo
2. at the same time, generate wrapper code for these model files
3. afterwards, execute arbitrary shell commands sequentially, if the KiCo compilation and wrapper code generation finished successfully

KiCo launch configurations work per project basis, thus every project has to create its own launch config. This is done automatically when performing `Right Click > Run As > KiCo Compilation` on a model file.

The **Run As** command will search for a KiCo launch config for the project. If there is such a config, the selected file is only added to the list of model files which should be compiled. If there is none, a launch config is created by using the main file and environment the project has been created with. If the main file and environment information could not be found, dialogs will query it from the user.

The **main file** of the launch config is used to set several file path variables, which can be used in several fields of the configuration, notably the shell commands to be executed, and wrapper code input. To use a variable the syntax is **${***variable_name***}**. The variables that are set are

- ▷ *main_name* : The file name, including its file extension (e.g. *MyModel.sct*)
- ▷ *main_path* : The project relative path (e.g. *src/MyModel.sct*)
- ▷ *main_loc* : The absolute file system path (e.g. */home/me/workspace/MyProject/src/MyModel.sct*)
- ▷ *main_name_no_ext* : The file name without its file extension (e.g. *MyModel*)

Further, similar variables for the *compiled* main file are set, that is, the main file in the directory of KIELER generated files (see below).

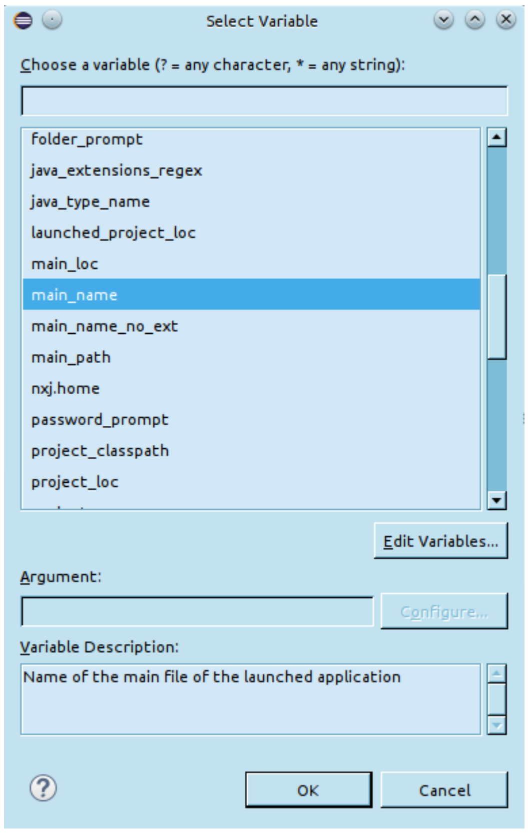
- ▷ *compiled_main_name* : The file name, including its file extension (e.g. *MyModel.sct*)
- ▷ *compiled_main_path* : The project relative path (e.g. *kieler-gen/MyModel.sct*)
- ▷ *compiled_main_loc* : The absolute file system path (e.g. */home/me/workspace/MyProject/kieler-gen/MyModel.sct*)
- ▷ *compiled_main_name_no_ext* : The file name without its file extension (e.g. *MyModel*)

Note: The variables are created in the first KiCo launch. So if you want to select them in a variable selection dialog of Eclipse, you must have started at least one KiCo launch configuration.

The values of the launch config can also be **(re)set to an environment**. This will revert the fields for the compilation target, wrapper code generation and command execution.

The **compilation via KiCo** is configured on the *Compilation* tab. Here you can add/remove files that should be compiled via KiCo and the target language as well as the file extension for the language (such as *.java* for Java). The files will be compiled sequentially in order of appearance in the list. Further, it is possible to add a path to a template file. This is useful to add surrounding content to the KiCo output. The placeholder *${kico_code}* can be used in the template.

On the *Execute* tab, a list of **shell commands** can be added. They are typically used to further compile the KiCo and wrapper code output and afterwards deploy the result to the target platform. The commands are executed sequentially in order as they appear in the list after the KiCo compilation and wrapper code generation finished successfully. If a


Select Variable

Choose a variable (? = any character, * = any string):

folder_prompt
java_extensions_regex
java_type_name
launched_project_loc
main_loc
main_name
main_name_no_ext
main_path
nxj.home
password_prompt
project_classpath
project_loc

Edit Variables...

Argument:

Configure...

Variable Description:

Name of the main file of the launched application

OK Cancel

Figure B.1. Variable selection dialog in Eclipse

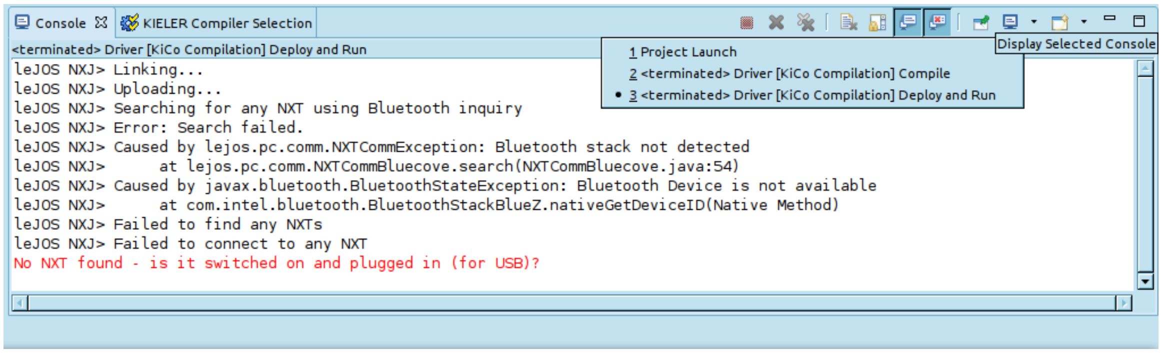
command fails (returns a non-zero exit code), following commands will not be excuted. The name of commands have to be unique and must not contain a comma.

The standard streams of executed shell commands (stdin, stderr, stdout), as well as errors from the KiCo compilation and wrapper code generation, are printed to the **Console View**.

B.2.1 Launch Groups

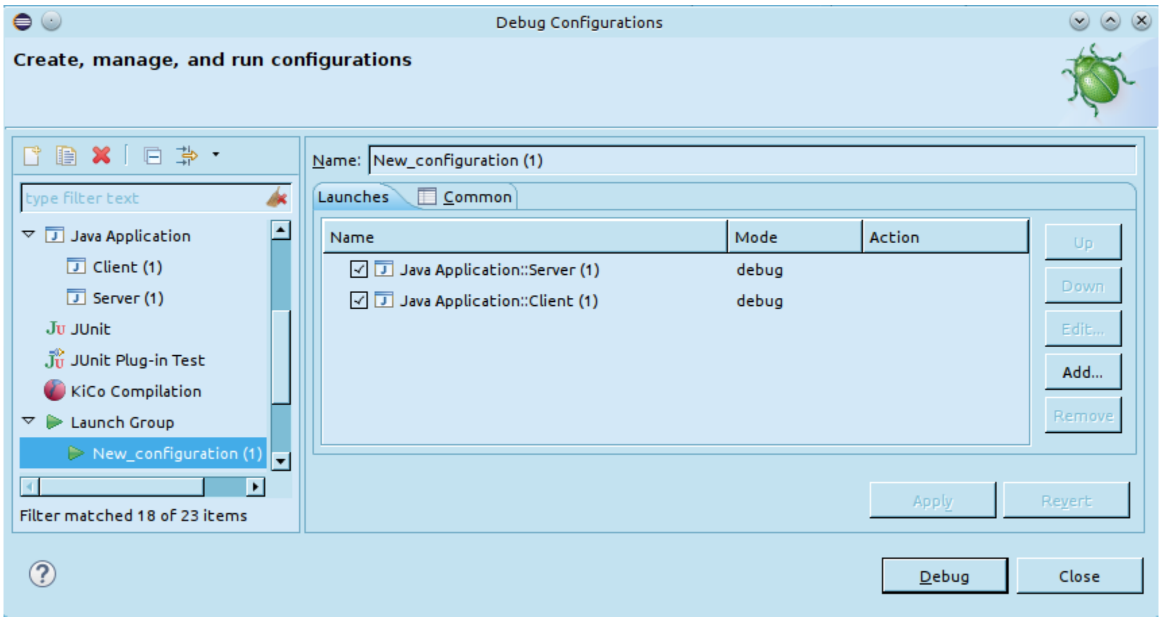
The list of shell commands are a simple mechanism to further compile and deploy code via command line tools. However, there are cases in which command line tools are not available or reasonable to use, for example because a different Eclipse launch configuration does a better job.

In this case it is desirable that the KiCo launch config only compiles the model and another Eclipse launch config does the rest. This can be achieved via *launch groups*. They let you define a launch configuration, which starts other launch configurations sequentially.


Console
KIELER Compiler Selection
Display Selected Console
1 Project Launch
2 <terminated> Driver [KiCo Compilation] Compile
3 <terminated> Driver [KiCo Compilation] Deploy and Run

```
<terminated> Driver [KiCo Compilation] Deploy and Run
leJOS NXJ> Linking...
leJOS NXJ> Uploading...
leJOS NXJ> Searching for any NXT using Bluetooth inquiry
leJOS NXJ> Error: Search failed.
leJOS NXJ> Caused by lejos.pc.comm.NXTCommException: Bluetooth stack not detected
leJOS NXJ>      at lejos.pc.comm.NXTCommBluecove.search(NXTCommBluecove.java:54)
leJOS NXJ> Caused by javax.bluetooth.BluetoothStateException: Bluetooth Device is not available
leJOS NXJ>      at com.intel.bluetooth.BluetoothStackBlueZ.nativeGetDeviceID(Native Method)
leJOS NXJ> Failed to find any NXTs
leJOS NXJ> Failed to connect to any NXT
No NXT found - is it switched on and plugged in (for USB)?
```

Figure B.2. Console View in Eclipse



Name: New_configuration (1)

Name	Mode	Action
☑ Java Application::Server (1)	debug	
☑ Java Application::Client (1)	debug	

Figure B.3. Launch group Example in Eclipse

To illustrate this, another use-case for launch groups is that you have a client-server application and want to start the client right after the server for debugging. Then you can create a launch config for the server and a launch config for the client. Afterwards you create a launch group to start these sequentially.

Launch groups are a part of the C/C++ Development Toolkit (CDT), although they provide a general mechanism that could be a part of any Eclipse IDE. The CDT is available in the Eclipse Marketplace (`Help > Eclipse Marketplace`)

B.3 Prom Environments

Environments are used to provide default settings for project creation and launch. They are configured in the preferences (`Window > Preferences > KIELER > Environments`).

An environment consists of

1. a unique **name**, that does not contain a comma
2. a **related project wizard**
3. information about a **main file** for the project
4. information about the **target code** KiCo should produce
5. information for **wrapper code generation**
6. a list of **shell commands** which should be run as part of a project launch

Besides the name, all of these are optional, but can improve the workflow. The related project wizard is run as part of the Prom project wizard and takes care of the actual project creation.

A main file typically contains the entry point of the program on the target environment. Its wrapper code initializes and runs the model and sets inputs and outputs to the physical components of the target device. To ease the project setup and because wrapper code for a specific target platform is often similar, it is possible to define default content for the main file. Therefore the field **main file origin** can contain an absolute file path to a file with the default contents of a newly created main file for this environment. Furthermore, predefined wrapper code snippets can be injected as part of a project launch, which is described below.

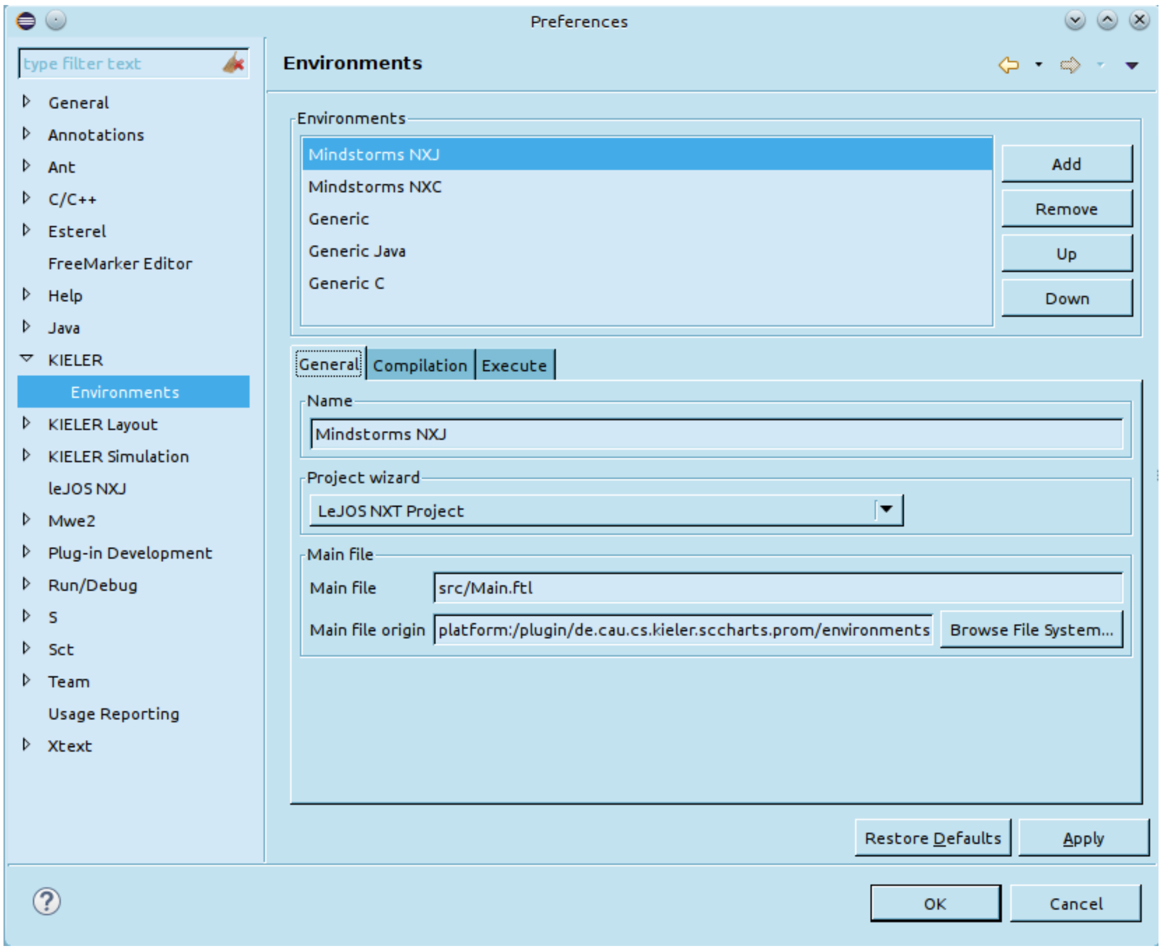
The **snippets origin** is used to initialize the wrapper code snippet directory of a newly created project.

The other fields are default settings for KiCo launch configurations.

B.3.1 Paths for Initial Content

The path for the main file origin accept an **absolute** file **path** as well as an **URL** with the platform protocol of Eclipse. An URL for the field has the form *plaftorm:/plugin/a.plugin.name/folder/in/the/plugin/file.txt*

The snippets origin works analog. It accepts an absolute directory path as well as an URL with the platform protocol which points to a directory. An URL for the field has the form *plaftorm:/plugin/a.plugin.name/folder/in/the/plugin*



Environments

Environments
Mindstorms NXJ
Mindstorms NXC
Generic
Generic Java
Generic C

Name
Mindstorms NXJ

Project wizard
LeJOS NXT Project

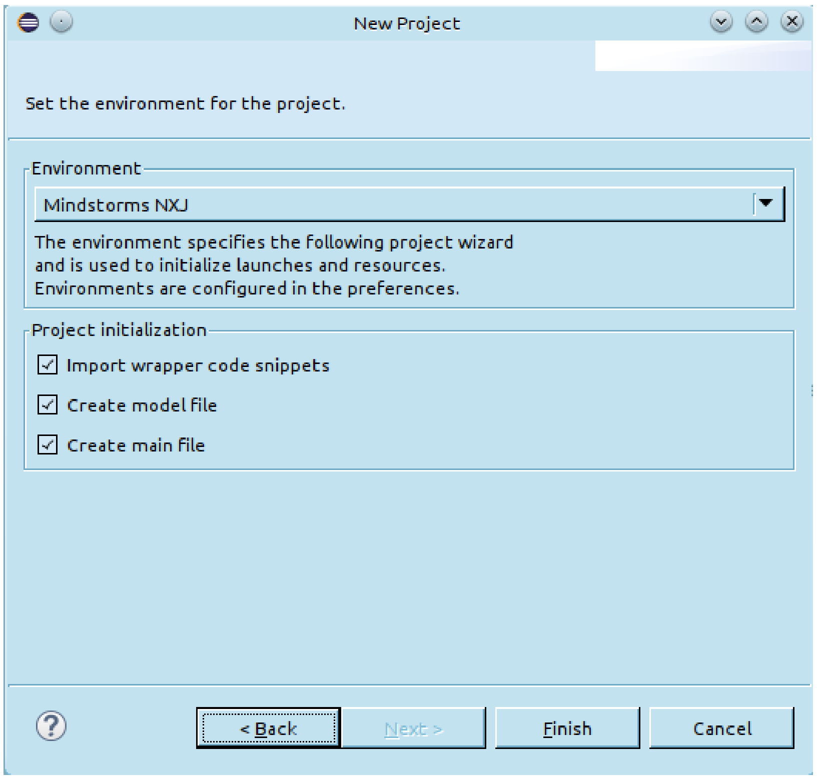
Main file
Main file: src/Main.ftl
Main file origin: platform:/plugin/de.cau.cs.kieler.sccharts.prom/environments

Figure B.4. Environments preferences page

B.4 Project Wizards with Prom

Prom provides project wizards, which can **create and initialize a project** with a **model file**, a **main file** and wrapper code **snippets**. The wizards for different model file types (e.g. SCChart project vs. Esterel project) differ only in the initial content for these. Other initial content is choosen from the environment, which is selected on the first page of a Prom wizard. The project creation itself is done by another wizard, that is started from within the Prom wizard.

If the *snippets directory* of an environment is a project relative path, the contents from the snippets origin will be copied to this location. If it is an absolute path, it is not copied to the project. Keeping snippets in a single, project indepentent folder, makes it easier to maintain them. For example it is possible to set an absolute path to a directory outside any


New Project

Set the environment for the project.

Environment: Mindstorms NXJ

The environment specifies the following project wizard
and is used to initialize launches and resources.
Environments are configured in the preferences.

Project initialization
- ☑ Import wrapper code snippets
- ☑ Create model file
- ☑ Create main file

Figure B.5. Prom project wizard

project as directory for wrapper code snippets. This directory can then be easily maintained using a version control system. Furthermore, if an issue occurs, it has to be addressed only once, because the snippets are not copied to every new project.

For example to create a project to develop Minstorms running leJOS, one can choose the SCCharts project wizard. In this wizard, one can choose the Mindstorms NXJ environment and define what will be initialized in the project (model file, main file, snippets). Now when pressing the finish button, the related project wizard from the leJOS plugin will be started. When it finishes, the newly created project is initialized with an initial model file, main file and wrapper code snippets.

B.5 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.

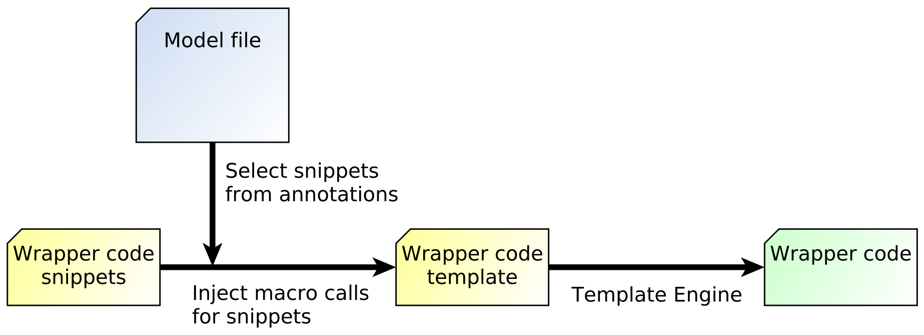


Figure B.6. Wrapper code generation scheme

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

In SCT files, annotations are added with an at-sign e.g. `@LightSensor S3`. You can use 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 e.g. because of a typo.

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

▷ **${model_name}** will be replaced with the last name of the compiled models.

▷ **${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.

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 language, 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`

```
...
// Initialization
${inits}
...
// Tick loop
while(...){
   // Input snippets
   ${inputs}
   // Reaction of model
   scchart.tick();
   //Output snippets
   ${outputs}
}
...
```

Figure B.7. Template file structure

B.5.1 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 a **FreeMarker IDE** as part of the JBoss Tools Project. It can be installed using the Eclipse Marketplace.

B.6 Automatically generated files

Files created by Prom are saved in the directory **kieler-gen**. Thereby the directory structure of files is retained, but without a starting Java source folder. This is because kieler-gen itself is a Java source folder.

For example (if *code* is not a Java source folder) the file *code/subfolder/MyModel.sct*, will be saved to *kieler-gen/code/subfolder/MyModel.sct*.

In contrast (if *src* is a Java source folder) the file *src/subfolder/MyModel.sct*, will be saved to *kieler-gen/subfolder/MyModel.sct*.

Bibliography

[And96] Charles André. *SyncCharts: A visual representation of reactive behaviors*. Tech. rep. RR 95–52, rev. RR 96–56. Sophia-Antipolis, France: I3S, Rev. April 1996.

[BC84] Gérard Berry and Laurent Cosserat. "The ESTEREL Synchronous Programming Language and its Mathematical Semantics". In: *Seminar on Concurrency, Carnegie-Mellon University*. Vol. 197. LNCS. Springer-Verlag, 1984, pp. 389–448. ISBN: 3-540-15670-4.

[FH10] Hauke Fuhrmann and Reinhard von Hanxleden. *Taming graphical modeling*. Technical Report 1003. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2010.

[FKB01] Duane K. Fields, Mark A. Kolb, and Shawn Bayern. *Web Development with JavaServer Pages*. Greenwich, CT, USA: Manning Publications Co., 2001. ISBN: 193011012X.

[HCR+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The synchronous data-flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. *SCCharts: Sequentially Constructive Statecharts for safety-critical applications*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.

[MSH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. "Compiling SCCharts—A case-study on interactive model-based compilation". In: *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2014, pp. 443–462. DOI: 10.1007/978-3-662-45234-9.

[MVN06] Dragos-Anton Manolescu, Markus Voelter, and James Noble. *Pattern Languages of Program Design*. 5th ed. Addison-Wesley Professional, 2006. ISBN: 9780321321947.

[Pto14] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL: http://ptolemy.org/books/Systems.