Supplementary Material for the Evaluation of the Publication – A Layered Reference Architecture for Model-based Quality Analysis

by Sandro Koch\textsuperscript{1}, Robert Heinrich\textsuperscript{1}, Ralf Reussner\textsuperscript{1}
facultative

(1) Information on affiliation as follows:
1 KASTEL - Institute of Information Security and Dependability

(2) Information on parallel publications in anthologies, conferences

(3) Details of projects, clients, sponsors, URLs, numbers and other project-related information is not entitled

This work was supported by the Federal Ministry of Education and Research (BMBF) under the funding number 01IS18067D, and the KASTEL institutional funding.

facultative

Institute of Information Security and Dependability (KASTEL), Prof. Ralf Reussner
Am Fasanengarten 5
76131 Karlsruhe
Germany

https://dsis.kastel.kit.edu/

Impressum

Karlsruher Institut für Technologie (KIT)
www.kit.edu

This document is licensed under the Creative Commons Attribution – Share Alike 4.0 International License (CC BY-SA 4.0): https://creativecommons.org/licenses/by-sa/4.0/deed.en

2022

ISSN: 2194-1629
Supplementary Material for the Evaluation of the Publication – A Layered Reference Architecture for Model-based Quality Analysis

Technical Report

Sandro Koch, Robert Heinrich, Ralf Reussner
{sandro.koch|heinrich|reussner}@kit.edu

01.06.2022
Contents

1. Introduction ........................................................................................................... 1

2. Case Studies ........................................................................................................... 3
       2.1.1. SimuLizar Overview ................................................................................. 3
       2.1.2. SimuLizar Refactoring ........................................................................... 4
       2.1.3. Modular SimuLizar (mSimuLizar) .............................................................. 4
   2.2. Business Process Simulation – Camunda ............................................................. 8
       2.2.1. Camunda Overview ................................................................................. 8
       2.2.2. Camunda Refactoring ............................................................................. 8
   2.3. Change Propagation Analysis – KAMP and KAMP4aPS ..................................... 10
       2.3.1. KAMP4APS Overview ........................................................................... 10
       2.3.2. KAMP4APS Refactoring ....................................................................... 10
   2.4. Energy Network Simulation – SmartGrid ............................................................ 11
       2.4.1. SmartGrid Overview .............................................................................. 11
       2.4.2. SmartGrid Refactoring .......................................................................... 11

3. Evaluation Tooling .................................................................................................... 13
   3.1. RefactorLizar – Evaluation Library .................................................................... 13
   3.2. RefactorLizar – Analysis Library ....................................................................... 13
       3.2.1. Feature Scatter identification .................................................................. 13
       3.2.2. Language Blob identification .................................................................. 13
       3.2.3. Identification of layer violations ................................................................. 14
       3.2.4. Identification of dependency cycles ......................................................... 14
   3.3. RefactorLizar – Refactoring Library .................................................................. 14
   3.4. RefactorLizar – Reference implementation ........................................................ 14
       3.4.1. Commands ............................................................................................... 14

4. Evaluation Data ........................................................................................................ 17
   4.1. Evolution Scenarios ......................................................................................... 17
       4.1.1. SimuLizar .................................................................................................. 17
       4.1.2. Camunda .................................................................................................. 20
       4.1.3. KAMP4aPS .............................................................................................. 22
       4.1.4. SmartGrid ............................................................................................... 24

Bibliography ................................................................................................................. 27
A. Appendix

A.1. Camunda

A.2. KAMP4aPS

A.3. SmartGrid
1. Introduction

In this technical report, we present the supplementary information for the evaluation of the Layered Reference Architecture for Model-based Quality Analysis. In chapter 2, we present the case studies, first in their monolithic and then in their modular form. We present installation instructions for the tools required to reproduce the evaluation results in chapter 3. In chapter 4, we provide detailed information about the evolution scenarios. The tooling and the results of the scenarios can be found online [7].
2. Case Studies

In this section, we present the four case studies we modularized according to our reference architecture for model-based analyses. The model-based analyses we used are SimuLizar, Camunda, KAMP4aPS, and SmartGrid. We built modular versions of the scenarios we extracted from the case study model-based analyses for the evaluation and we only refactored them according to the reference architecture’s guidelines. We did not fix bad smells that the reference design does not address because doing so would jeopardize the evaluation’s internal validity.

2.1. Software Architecture Quality Prediction – Palladio Simulator SimuLizar

The Palladio Simulator is an established software architecture quality analysis tool based on the Palladio Component Model (PCM). Contrary to its name, the Palladio Simulator consists of three performance analyses capable of determining the performance of software architecture: SimuCom, EventSim, and SimuLizar. Each of these analyses has a distinct set of features with different priorities. SimuCom covers most features of the PCM, it generates the analysis code based on the model, but it has performance issues for large software architectures. EventSim interprets instances of the PCM, and only supports the performance analyses of software architecture while ignoring many features of the PCM. In contrast to SimuCom, EventSim has fewer issues with large software architectures due to its event-based nature [8]. SimuLizar interprets the PCM, and it supports most of the PCM features. Due to their different approaches, their source code is not interchangeable; thus, the three analyses are incompatible. We focus on SimuLizar, as it is actively maintained. One of the main issues the developers had before the maintenance and development stopped were that changes in the PCM required changes in all three analyses. All three are historically grown model-based analyses, with the typical deterioration of the internal quality over time. SimuLizar is a historically grown model-based analysis, with the typical deterioration of the internal quality over time. Other historically grown model-based analyses show similar problems. As the quality of the analysis deteriorated, more and more effort was required to sustain all three.

2.1.1. SimuLizar Overview

The Palladio Simulator consists of three analyses (SimuLizar, SimuCom, and EventSim), each of which employs a distinct analysis approach and can make performance predictions based on the PCM. SimuLizar is the most sophisticated of the three analyses; thus, we have selected it for our case study. SimuLizar is developed since 2013; it is written in the
programming language Java. SimuLizar consists of 75 packages, 306 classes, 69 interfaces, and three enums; it is divided into 36 java-projects. SimuLizar has doubled in size since 2015, with classes increasing from around 150 to over 300. It also has a long history of evolutionary changes. SimuLizar features ten openly available extensions\(^1\) and many extensions that are not fully disclosed (e.g., student theses, experimental extensions). SimuLizar represents a historically grown and versatile model-based analysis that can analyse multiple aspects of software quality. If not stated otherwise, when we mention the term \textit{component} we refer to \textit{analysis component}, and when we mention the term \textit{feature} we refer to \textit{analysis feature}. Before the refactoring of SimuLizar, all dependencies on the metamodel PCM were consolidated in one analysis component, see fig. 2.1. We exclude the components that have no representation in the PCM due to the size of SimuLizar.

### 2.1.2. SimuLizar Refactoring

We started the modularisation with the release version 4.3 of the Palladio-Simulator, and used the modularised PCM presented in \([4, 10]\). Before we modularised SimuLizar, we had to change the dependencies of SimuLizar on the modular PCM. Changing the dependencies is necessary, as the modular PCM is not used in the Palladio-Simulator. After changing the dependencies, we analysed SimuLizar regarding the bad smells of \textit{Language Blob} and \textit{Feature Scatter}. We used the Language Blob bad smell to identify which classes we have to separate the components into the three desired layers. The Feature Scatter smell indicates which classes and components could be merged, as the refactoring of the Language Blobs results in many small classes. The Language Blob analysis resulted in 18 occurrences, and the Feature Scatter analysis resulted in 33 occurrences. First, we focused on the language blobs of components that are supposed to be on different layers. Therefore, we applied a horizontal-split refactoring to separate the analysis component in the layers \(\Pi\), \(\Delta\), and \(\Omega\), which resulted in three components. Then, we applied vertical-split refactorings to the three layers to separate the language blobs still present on these layers. The final step was to merge the components where the language features were scattered over different classes and components. We could not fix all occurrences of the Feature Scatter bad smell; for certain analysis operations, multiple language features are required. The model observing part of SimuLizar requires the \textit{modelobserver language feature} and the \textit{software usage language feature}. This resulted in nine components on \(\Pi\), 22 components on \(\Delta\), and one component on \(\Omega\). The component count increased from one component to 32 components. We reduced the number of Language Blobs from 18 to zero, and the number of Feature Scatters from 33 to ten. In the following sections 2.1.3.1 and 2.1.3.2 we present detailed information about the modular structure of SimuLizar after the refactoring.

### 2.1.3. Modular SimuLizar (mSimuLizar)

Figure 2.2 depicts the structure of SimuLizar after the modularisation. In the figure, we exclude the analysis components that have no representation in the language, e.g. events, the interpreter component, or the reconfiguration component, as most analysis

---

\(^1\) https://sdqweb.ipd.kit.edu/wiki/SimuLizar
components have dependencies on them. Including these additional components renders the already complex figure unintelligible.

2.1.3.1. Paradigm

**Composition:** The composition component handles the assembly of resources of the PCM. On the paradigm layer, the functionality of the composition component is prepared to handle any type of resources. The assembly of component types includes the preparation of resources. Preparing a resource means, setting the context and the context hierarchy of the resource. The composition component provides functionality for adding or deleting a resource and it also provides the connectors required to compose resources.

**Constants:** The constants component provides the constants required throughout the analysis of PCM instances.

**Repository:** The repository component on the paradigm layer manages the roles defined in the PCM. The PCM defines required and provided roles for components. In this component, the roles, e.g. provided and required roles, are managed. It provides interfaces to receive these roles, and also it provides interfaces to receive the signatures defined in the PCM. The main portion of the repository component is the repository switch. The switch contains the interpretation of the roles. It also contains the analysis code concerning the required and provided roles. The signatures are implicitly used throughout the analysis code.

**Runtimetype:** The runtimetype component provides abstract classes and interfaces for managing the state of the analysis. It holds the PCM instance, the event notification helper, and a registry of the analysed components. The component registry is an interface for validating whether a component is available for the analysis. It also provides add and fetch operations for the PCM components. The event notification helper is an interface for firing events and removing listeners.

**Seff:** The Service Effect Specification (SEFF) in the PCM represents the basic actions of a component. The seff component provides the interpretation and the analysis code for the elements of the seff language feature of the PCM. The seff component contains the interpreter for the seff types. For each seff type, the seff component contains the analysis code required for the elements.

**Usage:** The usage component provides the handling of probabilities defined in the usage language feature of the PCM. Probabilities are required when the analysis encounters a
branch. The usage component determines in which direction the analysis must proceed. Besides branches, the usage component also provides the scheduling of delays. Another part of the usage component is the handling of loops. Based on the size of a loop, it determines the time required to finish the loop. Furthermore, the usage component provides an interface to manage user actions.

Variables: The variables component provides the evaluation of the model instance. It creates an evaluator instance containing the variable characterisation of the PCM and the model evaluator. The evaluation provides a condition checker, which checks whether a boolean expression in a condition holds. The variable component also provides the generation of random variables.

2.1.3.2. Domain

Behaviour seff: The behaviour seff component provides the analysis code for the PCM model elements external call action, acquire action, collection iterator action, set variable action, and release action. The analysis code requires information about the infrastructure; thus, in this component, remain dependencies on the infrastructure language feature. The
behaviour self component also provides analysis code which determines probabilistic transitions when encountering branches.

**Domain repository:** The *domain repository* component provides an interface for implementing the analysis code for the PCM model elements *provided role* and *signature*.

**Infrastructure composition:** The *infrastructure composition* component provides the analysis code for the PCM model elements *assembly infrastructure connector* and *required infrastructure delegation connector*. The component utilises the composition and repository component of the π layer.

**Modelobserver:** The *modelobserver* component provides the analysis code for the PCM model elements *communication link resource specification*, *linking resource*, *processing resource specification*, *resource container*, *workload*, *closed workload*, *open workload*, and *usage scenario*. The component requires, in addition to the modelobserver language feature, the *software usage* language feature, thus it holds dependencies on PCM types of these two language features.

**Modelobserver environment:** The *modelobserver environment* component provides the analysis code for the PCM model element *resource environment*. This component handles the modelobserver component, and it provides observers for the said model and the resource environment.

**Notification:** The *notification* component provides the analysis code for the PCM model elements *operation provided role*, *operation signature*, *external call action*, *entry level system call*, and *usage scenario*. This component has dependencies on four language features to perform the analysis.

**Runtimestate:** The *runtimestate* component provides the analysis code for the PCM model elements *resource environment*, and *assembly context*. The runtimestate component has only two dependencies on two language features, but it consolidates the state of the analysed system. It utilises direct knowledge (i.e., usage model component), or it utilises the modelobserver component to manage the runtime state of the analysis.

**Simulated component:** The *simulated component* provides the analysis code for the PCM model element *passive resource*. It represents two types of components mSimuLizar can analyse. The first component is a basic component that can be monitored, and it can acquire and release resources. The second component is a composite component, consisting of a set of basic components.

**Software composition:** The *software composition* component provides the analysis code for the PCM model elements *assembly connector*, *required delegation connector*, and *composite component*.

**Software repository:** The *software repository* component provides the analysis code for the PCM model elements *basic component* and *service effect specification*.

**Software usage:** The *software usage* component provides the analysis code for the PCM model elements *entry level system call*, *usage scenario*, and *usage switch*.

**Usage model:** The *simulated component* provides the analysis code for the PCM model elements *usage model*, *usage scenario*, *workload*, *closed workload*, *open workload*, *software usage package*. 
2.2. Business Process Simulation – Camunda

The analysis Camunda is a workflow and simulation engine based on the Business Process Modelling Notation 2 (BPMN2) Domain-Specific Modelling Language (DSML). The BPMN2 is developed by the Object Management Group (OMG). It is also an International Organization for Standardization (ISO) standard for modelling business processes. We selected Camunda as a case study because it covers the additional domain of business process analysis, and it can be used for further refactorings since, besides the standard BPMN2, it also supports the Case Management Model and Notation (CMMN 1.1) and the Decision Model Notation (DMN 1.1). Camunda is a fork of the free workflow management system Activiti, developed in 2010. In 2013 Camunda BPM was forked from Activiti as an open-source project by the company Camunda in Berlin. Our refactorings focus on the Camunda BPM Platform, which consolidates the dependencies on the metamodel. Due to the size of the Camunda BPM Platform\(^2\) (over 500,000 lines of code), we were unable to refactor it in a reasonable time frame; therefore, we focused our refactorings on the affected analysis components and files of our scenarios.

2.2.1. Camunda Overview

The Camunda BMN Platform consists of 15 modules that also contain modules. It has 52 modules in total. The model-api module consolidates the dependencies on the BPMN2 metamodel. Figure 2.3 depicts the internal dependency structure of the Camunda BPM Platform. Turquoise nodes represent dependencies on org.camunda.bpm modules. Purple nodes represent dependencies on org.camunda.bpm.model modules. Black nodes represent dependencies on the remaining org.camunda modules.

2.2.2. Camunda Refactoring

Before we could refactor the Camunda BPM Platform, we had to adapt the dependencies of the analysis code to the modular BPMN2 DSML \[^4, 10\]. The turquoise nodes in fig. 2.3 are the modules that had to be modified. The dependencies of the Camunda BPM Platform regarding the mBPMN2 metamodel are similar to the structure shown in fig. 2.1. In the org.camunda.bpm.model module are the dependencies on the mBPMN2 metamodel consolidated. As we did not refactor the whole analysis, details regarding the refactoring will be presented in chapter 4.

\(^2\) https://github.com/camunda/camunda-bpm-platform
2.2. Business Process Simulation – Camunda

Figure 2.3.: Camunda BPM Platform Dependency Structure
2.3. Change Propagation Analysis – KAMP and KAMP4APS

The Karlsruhe Architecture Maintainability Prediction for Automated Production Systems (KAMP4APS) analysis is a single-purpose analysis; its metamodel is only used by a single analysis. Although the Karlsruhe Architecture Maintainability Prediction (KAMP)-Frameworks methodology [3] is built to support the domains of software systems [5], business processes [9], production systems [6], and Programmable Logic Controller (PLC) software [2], each domain requires a dedicated analysis and metamodel. Due to the selection of the KAMP4APS metamodel in our previous work, we will focus solely on the KAMP4APS analysis. KAMP4APS covers an additional domain, which extends the diversity of our case studies. The KAMP4APS metamodel and analysis has been under development since 2016; it contains six analysis components, one of which consolidates the dependencies on the metamodel.

2.3.1. KAMP4APS Overview

KAMP4APS has a generic part that provides the framework for the change impact analysis. This framework is the foundation of the KAMP Methodology [3]. It provides a domain-independent part, consisting of a domain-independent modification metamodel, a set of algorithms to derive a task list, eliminate duplicates and sort elements in the task list. It also provides a metamodel and algorithms to support decision-making regarding changes in the analysed domain. Based on the domain-independent part, each domain has to provide a metamodel of the domain that will be analysed. The change impact analysis requires a structural metamodel and a non-structural metamodel. In the context of KAMP4APS the structural parts of the metamodel are the electrical and mechanical parts in a production system. Non-structural parts are, for example, documentation, drawings or tests. These models are used in the rule engine of KAMP to determine the impact of changes to the structural and non-structural parts of the system. Although KAMP4APS is separated into a domain-independent and a domain-dependent part, and the further separation into models, algorithms, structural and non-structural elements, it consists of a module containing the domain-independent part (KAMP) and a model containing the domain-specific part (KAMP4APS).

2.3.2. KAMP4APS Refactoring

Before we could refactor KAMP4APS, we had to adapt the dependencies of the analysis code to the modular KAMP4APS metamodel [4][10]. The dependencies of the KAMP4APS regarding the modular KAMP4APS metamodel are like the structure shown in fig 2.1. The dependencies on the modular KAMP4APS metamodel are consolidated in the KAMP4APS module. As we did not refactor the whole analysis, details regarding the refactoring will be presented in chapter [4].
2.4. Energy Network Simulation – SmartGrid

As the KAMP4aPS analysis, the SmartGrid analysis is also a single-purpose analysis. The SmartGrid energy network simulation performs an impact and resilience analysis. The metamodel is used to model topologies of smart grid energy networks. It also adds the domain of energy network analysis to our case studies; it is the second-youngest analysis, the development started in 2014. Compared to the analysis SmartGrid, the size of the SmartGrid analysis is smaller by a factor of ten. The SmartGrid contains 15 analysis components, one of which consolidates the dependencies on the metamodel.

2.4.1. SmartGrid Overview

The Smart Grid Resilience Framework allows modelling and analyse critical infrastructures. With metamodel of this analysis, the topology of a smart grid can be modelled. The analysis allows for simulating cyberattacks; it also allows for determining the impact of such attacks on the infrastructure. These simulations can be coupled with a power load simulation and a simulation of critical infrastructures, which are developed by our research partners. In contrast to the previous case studies, the metamodel is integrated into the analysis.

2.4.2. SmartGrid Refactoring

Before we could refactor the SmartGrid analysis, we had to adapt the dependencies of the analysis code to the modular SmartGrid metamodel [4, 10]. The dependencies of the SmartGrid regarding the modular SmartGrid metamodel are like the structure shown in fig. 2.1. The dependencies on the modular SmartGrid metamodel are consolidated in the `smartgrid.attackersimulation` and the `smartgrid.impactanalysis` module. Although technically, these two modules represent two different analyses, we consider them as one. Each represents a analysis feature of the SmartGrid analysis. As we did not refactor the whole analysis, details regarding the refactoring will be presented in chapter 4.
3. Evaluation Tooling

We have developed a Java library to support the evaluation of our case studies. Besides the evaluation, the library also supports the analysis and refactoring of model-based analyses. The library supports Java- and EMF-based analyses. The library, called RefactorLizar\(^1\), serves three purposes. Determining cohesion, coupling, and complexity of model-based analyses, see section 3.1. Analysis of model-based analyses regarding reference architecture violations, see section 3.2. Automated refactoring of model-based analysis, see section 3.3.

3.1. RefactorLizar – Evaluation Library

RefactorLizar allows developers to determine cohesion, coupling, and complexity of model-based analyses using the hypergraph metrics of Allen et al. \([1]\). The evaluation of metamodels is not supported by RefactorLizar Evaluation Library.

3.2. RefactorLizar – Analysis Library

The analysis part of RefactorLizar provides information about a modularized model-based analysis. In order for the analysis to function, the developer must provide a DSML and a corresponding model-based analysis as input. RefactorLizar consists of the following four analyses: 3.2.1 Feature Scatter identification, 3.2.2 Language Blob identification, 3.2.3 Identification of layer violations, and 3.2.4 Identification of dependency cycles.

3.2.1. Feature Scatter identification

When multiple analysis components have dependencies on the same DSML language type, we define it as Feature Scatter. A Feature Scatter violates our reference architecture, as an analysis component should only depend on one language feature. Besides the multiple dependencies, the feature and its corresponding component must be located on a single layer. RefactorLizar can identify the scattering of features; it provides the developer with a list of components that depend on a single feature.

3.2.2. Language Blob identification

When analysis components have multiple dependencies on DSML language types, we define it as Language Blob. A Language Blob violates our reference architecture, as an analysis component should have only one DSML feature as dependency. RefactorLizar

\(^1\) https://github.com/MoSimEngine/RefactorLizar
supports the developer to identify components that have dependencies to more than one DSML feature. The analysis result provides the developer with a list of language features, a component depends on. The developer can define a threshold to set the minimal amount of dependencies before a component is added to the list.

### 3.2.3. Identification of layer violations

When a model-based analysis feature is on a different layer than its corresponding model-based analysis component, we define it as a layer violation. Also, when a model-based analysis feature is on a different layer than its corresponding DSML feature, we define it as a layer violation. RefactorLizar allows the developer to detect when dependencies between layers point in the wrong direction or surpass adjacent layers. The layer identification of analysis components requires further annotation by the analysis developer, while the layer identification of referenced DSML types is made automatically.

### 3.2.4. Identification of dependency cycles

A bidirectional dependency between analysis components is the simplest form of a dependency cycle. RefactorLizar can detect dependency cycles on class and component level.

### 3.3. RefactorLizar – Refactoring Library

RefactorLizar supports the following basic refactorings: Move type members, introduce inheritance, adapt interface extension, change the visibility of methods and attributes, delete classes, delete methods and attributes, and create new types. RefactorLizar can provide these refactorings automatically: Class Split, Class Merge, Breaking Dependency Cycle, Dependency Inversion, Horizontal Split, Vertical Split, Merge, and Extension Extraction.

### 3.4. RefactorLizar – Reference implementation

This project provides a command-line interface for the RefactorLizar library. We made a reference implementation to demonstrate the analysis features of RefactorLizar. The reference implementation is available on our GitHub page[2] and supplementary material[7]. The implementation is provided as a Command Line Interface (CLI) tool. We also plan to provide a visual interface for the RefactorLizar library as a Visual Studio Code extension.

#### 3.4.1. Commands

RefactorLizarCLI utilizes PicoCLI[3] and GraalVM[4]; thus, commands can be started via the provided binary or via `gradle run -args="<command/s>"`.

---


[3] https://picocli.info/

3.4. RefactorLizar – Reference implementation

3.4.1. evaluateCode

The command `evaluateCode` evaluates hypergraph code metrics for the given source path. The arguments `data-types` and `observed-system` is the path to file for ignored/included types. Every line in this file is seen as a regex tested against the qualified type names. The `data-types` parameter represents the ignored types and the `observed-system` parameter represents the included types. The `code` argument provides the path to the analysis to evaluate.

3.4.1.2. adaptDependencies

The command `adaptDependencies` changes imports of simulator code according to the new, modular metamodel. The command requires a CSV file that contains the mapping of the modular metamodel types mapped to the monolithic metamodel types. The argument `csv-path` provides the path to the CSV file. The argument `simulator-code` provides the path to the analysis.

3.4.1.3. findDependencyCycleSmell

The command `findDependencyCycleSmell` finds occurrences of the dependency cycle smell. The `analysis-level` argument sets the detail level of the result. Available analysis levels are type, component and package. The `language` argument is the path to the metamodel. The `simulator` argument is the path to the analysis code. If the code is eclipse-based the flag `input-type-eclipse` allows to handle eclipse-based analyses.

3.4.1.4. findDependencyDirectionSmell

Find occurrences of the dependency direction smell. Layers must be ordered from bottom to top and separated by `. `. Available analysis levels are type, component and package. The `analysis-level` argument sets the detail level of the result. Available analysis levels are type, component and package. The `language` argument is the path to the metamodel. The `simulator` argument is the path to the analysis code.

3.4.1.5. showTypesInMetamodels

The command `showTypesInMetamodels` lists all metamodel types. The argument `language-root` points to the root of the metamodel. The result can be used to determine the utilisation of a metamodel in a model-based analysis. The utilisation is the number of all types in a metamodel in relation to the types of a metamodel used in a model-based analysis.

3.4.1.6. findFeatureScatteringSmell

Find occurrences of the feature scattering smell. The `analysis-level` argument sets the detail level of the result. Available analysis levels are type, component and package. The `language` argument is the path to the metamodel. The `simulator` argument is the path to the analysis code. If the code is eclipse-based the flag `input-type-eclipse` allows to handle eclipse-based analyses.
3. Evaluation Tooling

3.4.1.7. findDependencyLayerSmell

Find occurrences of the improper simulator layering smell. Available analysis levels are type, component and package. The analysis-level argument sets the detail level of the result. Available analysis levels are type, component and package. The language argument is the path to the metamodel. The simulator argument is the path to the analysis code. If the code is eclipse-based the flag input-type-eclipse allows to handle eclipse-based analyses.

3.4.1.8. findLanguageBlobSmell

Find occurrences of the language blobs smell. Available analysis levels are type, component and package. The analysis-level argument sets the detail level of the result. Available analysis levels are type, component and package. The language argument is the path to the metamodel. The simulator argument is the path to the analysis code. If the code is eclipse-based the flag input-type-eclipse allows to handle eclipse-based analyses.
4. Evaluation Data

4.1. Evolution Scenarios

In this section, we present the evolution scenarios of our four case studies. We identified ten scenarios per case study, ergo 40 scenarios in total. Each case study provided historical evolution scenarios; we did not have to define potential or random evolution scenarios. Historical evolution scenarios can affect files without dependencies on the DSML; thus, we did not apply any refactoring to these files. Also, we did not consider these files when we calculated the metrics cohesion, coupling, and complexity. For each case study, we provide sources for the DSML and model-based analysis in their monolithic and modular state. The scenarios can also be found in our reproduction package [7]. To correctly identify the scenario in the source code, we provide a commit hash or revision number and the date when the commit occurred.

4.1.1. SimuLizar

For the model-based analysis SimuLizar, we identified ten historical evolution scenarios. The reproduction data for SimuLizar contains the refactored code of the model-based analysis. Table 4.1 contains links to the monolithic and modular model-based analysis.

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>PCM [10]</td>
<td>-</td>
</tr>
<tr>
<td>Modular Language</td>
<td>mPCM [10]</td>
<td>-</td>
</tr>
<tr>
<td>Analysis</td>
<td>SimuLizar</td>
<td>Palladio-Analyzer-SimuLizar [7] master: b6b69b4f1</td>
</tr>
<tr>
<td>Modular Analysis</td>
<td>mSimuLizar</td>
<td>mSimuLizar [7]</td>
</tr>
</tbody>
</table>

Table 4.1.: Overview SimuLizar Projects

4.1.1.1. Scenario 01 – RepositoryComponentSwitch uses Extensible RDSeffSwitches

The first scenario is the commit 7542134. The commit occurred on Monday, April 24th 2017. In the monolith, four files are changed. The following files are affected by the commit: RDSeffSwitch, RepositoryComponentSwitch, AbstractRDSeffSwitchFactory, and IComposableSwitch.

4. Evaluation Data

4.1.1.2. Scenario 02 – Deleted ModelAccess Class


4.1.1.3. Scenario 03 – Fix Project Structure - Migrate RDSeffSwitch to Tycho

The third scenario is the commit 02511a37. The commit occurred on Monday, July 30th 2018. In the monolith, three files are changed. The following files are affected by the commit: AbstractRDSeffSwitchFactory, ExplicitDispatchComposedSwitch, and IComposableSwitch.

4.1.1.4. Scenario 04 – Added Mechanism to Explicitly Switch Based on Superclass

The fourth scenario is the commit d973511. The commit occurred on Tuesday, December 12th 2017. In the monolith, three files are changed. The following files are affected by the commit: RepositoryComponentSwitch, AbstractRDSeffSwitchFactory, and ExplicitDispatchComposedSwitch.

4.1.1.5. Scenario 05 – Add Monitorrepository to Feature Dependencies

The fifth scenario is the revision r34181. The commit occurred on Monday, April 24th 2017. In the monolith, four files are changed. The following files are affected by the commit: AbstractRDSeffSwitchFactory, IComposableSwitch, RDSeffSwitch, and RepositoryComponentSwitch.

4.1.1.6. Scenario 06 – Fixed Metadata for the HDD Patch

The sixth scenario is the revision r33820. The commit occurred on Friday, November 11th 2016. In the monolith, one file is changed. The following file is affected by the commit: RDSeffSwitch.

4.1.1.7. Scenario 07 – Include New Aggregation Plugin into Simulizar Feature

The seventh scenario is the commit r32804. The commit occurred on Friday, August 5th 2016. In the monolith, six files are changed. The following files are affected by the commit: AbstractProbeFrameworkListener, PRMRecorder, AbstractModelObserver, ResourceEnvironmentSyncer, AbstractSimuLizarRuntimeState, and MonitorRepositoryUtil.
4.1. Evolution Scenarios

4.1.1.8. Scenario 08 – Only Record Runtime Measurements


4.1.1.9. Scenario 09 – Generalized Response Times Aggregator

The ninth scenario is the revision r32166. The commit occurred on Tuesday, May 31st 2016. In the monolith, four files are changed. The following files are affected by the commit: AbstractSimuLizarRuntimeState, ComponentInstanceRegistry, SimulatedBasicComponentInstance, and SimulatedComponentInstance.

4.1.1.10. Scenario 10 – Added Missing Reconfiguration Rule

The tenth scenario is the revision r31800. The commit occurred on Tuesday, April 19th 2016. In the monolith, six files are changed. The following files are affected by the commit: EventNotificationHelper, RepositoryComponentSwitch, AbstractInterpreterListener, AbstractRecordingProbeFrameworkListenerDecorator, AssemblyProvidedOperationPassedEvent, and IInterpreterListener.
4. Evaluation Data

4.1.2. Camunda

For the model-based analysis Camunda, we identified ten historical evolution scenarios. The reproduction data for Camunda contains the classes of the ten scenarios. Each scenario is divided into two folders, the classes of the monolithic version is contained in the folder before and the classes of the modular version after the refactoring is contained in the after folder. Table 4.2 contains links to the monolithic and modular model-based analysis.

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>BPMN/Camunda [10]</td>
<td>–</td>
</tr>
<tr>
<td>Modular Language</td>
<td>mBPMN/Camunda [10]</td>
<td>–</td>
</tr>
<tr>
<td>Analysis</td>
<td>Camunda</td>
<td>Camunda GitHub[2]</td>
</tr>
<tr>
<td>Modular Analysis</td>
<td>mCamunda</td>
<td>master:f5c2d559d</td>
</tr>
</tbody>
</table>

Table 4.2.: Overview Camunda Projects

4.1.2.1. Scenario 01 – Add Timeout Task Listener

The first scenario is the commit d53583a. The commit occurred on Wednesday, August 21st 2019. In the monolith, five files are changed. See fig. A.1 and fig. A.2. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractCatchEventBuilder, AbstractUserTaskBuilder, CamundaTaskListenerImpl, and CamundaTaskListener.

4.1.2.2. Scenario 02 – Introduce errorMessage for Error Definitions

The second scenario is the commit b129522. The commit occurred on Friday, July 5th 2019. In the monolith, six files are changed. See fig. A.3 and fig. A.4. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractBoundaryEventBuilder, AbstractEndEventBuilder, AbstractErrorEventDefinitionBuilder, AbstractStartEventBuilder, and BpmnModelConstants.

4.1.2.3. Scenario 03 – Add Variable Specification to Conditional Event

The third scenario is the commit 14ad97ae. The commit occurred on Wednesday, October 5th 2016. In the monolith, four files are changed. See fig. A.5 and fig. A.6. The following files are affected by the commit: AbstractConditionalEventDefinitionBuilder, BpmnModelConstants, ConditionalEventDefinitionImpl, and ConditionalEventDefinition.

4.1.2.4. Scenario 04 – Remove incrementalIntervals Property

The fourth scenario is the commit a337b8f6. The commit occurred on Friday, September 8th 2017. In the monolith, four files are changed. See fig. A.7 and fig. A.8. The following files are affected by the commit: Bpmn, AbstractFlowNodeBuilder, BpmnModelConstants, CamundaIncrementalIntervalsImpl, and CamundaIncrementalIntervals.

4.1.2.5. Scenario 05 – Set Marker to Visible for Exclusive Gateway

The fifth scenario is the commit 7cf3cdff. The commit occurred on Thursday, June 1st 2017. In the monolith, one file is changed. See fig. A.9 and fig. A.10. The following file is affected by the commit: AbstractFlowNodeBuilder.

4.1.2.6. Scenario 06 – Removed errorMessage Attribute in endErrorEvent

The sixth scenario is the commit 4a5d7bc7c. The commit occurred on Monday, June 6th 2016. In the monolith, eight files are changed. See fig. A.11 and fig. A.12. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractBoundaryEventBuilder, AbstractEndEventBuilder, AbstractErrorEventDefinitionBuilder, AbstractStartEventBuilder, BpmnModelConstants, ErrorImpl, and Error.

4.1.2.7. Scenario 07 – Added Error Definition Variables

The seventh scenario is the commit 31e9a1324. The commit occurred on Thursday, June 2nd 2016. In the monolith, eleven files are changed. See fig. A.13 and fig. A.14. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractBoundaryEventBuilder, AbstractEndEventBuilder, AbstractErrorEventDefinitionBuilder, AbstractStartEventBuilder, ErrorEventDefinitionBuilder, BpmnModelConstants, ErrorEventDefinitionImpl, ErrorImpl, Error, and ErrorEventDefinition.

4.1.2.8. Scenario 08 – Add Convenience Methods to Allow Using Classes Instead

The eighth scenario is the commit 1d2a508c. The commit occurred on Friday, March 24th 2017. In the monolith, six files are changed. See fig. A.15 and fig. A.16. The following files are affected by the commit: AbstractBusinessRuleTaskBuilder, AbstractCallActivityBuilder, AbstractFlowNodeBuilder, AbstractSendTaskBuilder, AbstractServiceTaskBuilder, and AbstractUserTaskBuilder.

4.1.2.9. Scenario 09 – Create and Reference Message with the Fluent Builder

The ninth scenario is the commit 677b3c6. The commit occurred on Monday, February 1st 2016. In the monolith, six files are changed. See fig. A.17 and fig. A.18. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractCatchEventBuilder, AbstractFlowNodeBuilder, AbstractReceiveTaskBuilder, AbstractSendTaskBuilder, and AbstractThrowEventBuilder.

4.1.2.10. Scenario 10 – Add Support for camunda:connector Extension Element

The tenth scenario is the commit c30dbc8e. The commit occurred on Tuesday, August 5th 2014. In the monolith, six files are changed. See fig. A.19 and fig. A.20. The following files are affected by the commit: Bpmn, BpmnModelConstants, CamundaConnectorIdImpl, CamundaConnectorImpl, CamundaConnector, and CamundaConnectorId.
4. Evaluation Data

4.1.3. KAMP4aPS

For the model-based analysis KAMP4aPS, we identified ten historical evolution scenarios. The reproduction data for KAMP4aPS contains the classes of the ten scenarios. Each scenario is divided into two folders, the classes of the monolithic version is contained in the folder before and the classes of the modular version after the refactoring is contained in the after folder. Table 4.3 contains links to the monolithic and modular model-based analysis.

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>KAMP4aPS Lang [10]</td>
<td></td>
</tr>
<tr>
<td>Modular Language</td>
<td>mKAMP4aPS Lang [10]</td>
<td></td>
</tr>
<tr>
<td>Analysis</td>
<td>KAMP4aPS GitHub</td>
<td>master: HEAD</td>
</tr>
<tr>
<td>Modular Analysis</td>
<td>mKAMP4aPS</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3.: Overview KAMP4aPS Projects

4.1.3.1. Scenario 01 – Add Lookup for Interface Elements

The scenario is the commit 3126580b. The commit occurred on Sunday, March 19th 2017. In the monolith, four files are changed. See fig. A.21 and fig. A.22. The following files are affected by the commit: ArchitectureAnnotationLookup, AbstractKAPSDifferenceCalculation, AbstractKAPSEnrichedWorkplanDerivation, and SwitchChanges.

4.1.3.2. Scenario 02 – Add Super Type to Mechanical Assembly

The scenario is the commit 2d37dc02. The commit occurred on Monday, October 23rd 2017. In the monolith, four files are changed. See fig. A.23 and fig. A.24. The following files are affected by the commit: APSArchitectureModelLookup, ModuleChanges, MicroSwitchModuleChange, and RampChange.

4.1.3.3. Scenario 03 – Add Class for Micro Switch Change

The scenario is the commit c17f986e5. The commit occurred on Friday, August 18th 2017. In the monolith, six files are changed. See fig. A.25 and fig. A.26. The following files are affected by the commit: APSArchitectureModelLookup, APSSubactivityDerivation, MicroSwitchModuleChange, SwitchChanges, and LabelCustomizing.

4.1.3.4. Scenario 04 – Add Meta Class for Change

The scenario is the commit 1f78d0c0. The commit occurred on Friday, August 18th 2017. In the monolith, ten files are changed. See fig. A.27 and fig. A.28. The following files are affected by the commit: Change, ComponentChanges, InterfaceChanges, ModuleChanges,
4.1. Evolution Scenarios

StructureChanges, BusChanges, RampChange, SensorChanges, SignalInterfacePropagation, and SwitchChanges.

4.1.3.5. Scenario 05 – Update Ramp Change Scenario

The scenario is the commit 3f5acd29. The commit occurred on Monday, May 14th 2018. In the monolith, two files are changed. See fig. A.29 and fig. A.30. The following files are affected by the commit: APSChangePropagationAnalysis, and RampChange.

4.1.3.6. Scenario 06 – Refactoring Names of Change Classes

The scenario is the commit 8491dd9b. The commit occurred on Tuesday, August 15th 2017. In the monolith, four files are changed. See fig. A.31 and fig. A.32. The following files are affected by the commit: BusChanges, SensorChanges, SignalInterfacePropagation, and SwitchChanges.

4.1.3.7. Scenario 07 – Introduce HMI

The scenario is the commit d54511fe. The commit occurred on Thursday, April 26th 2018. In the monolith, six files are changed. See fig. A.33 and fig. A.34. The following files are affected by the commit: APSArchitectureVersion, APSArchitectureVersionPersistency, APSChangePropagationAnalysis, and APSDifferenceCalculation.

4.1.3.8. Scenario 08 – Adapt Change Propagation Analysis Regarding PLC Entry Points

The scenario is the commit 5dae880b. The commit occurred on Tuesday, February 27th 2018. In the monolith, five files are changed. See fig. A.35 and fig. A.36. The following files are affected by the commit: APSArchitectureModelFactoryFacade, APSArchitectureVersion, APSArchitectureVersionPersistency, APSChangePropagationAnalysis, and InterfaceChanges.

4.1.3.9. Scenario 09 – Introduce Duplicate Removal

The scenario is the commit a5dcc00c. The commit occurred on Wednesday, January 11th 2017. In the monolith, five files are changed. See fig. A.37 and fig. A.38. The following files are affected by the commit: AbstractKAPSChangePropagationAnalysis, ArchitectureAnnotationLookup, ArchitectureModelLookup, ArchitectureVersion, and ArchitectureVersionPersistency.

4.1.3.10. Scenario 10 – Refactor Function Names and Introduce Version

The scenario is the commit 1d2a508c. The commit occurred on Wednesday, January 11th 2017. In the monolith, three files are changed. See fig. A.39 and fig. A.40. The following files are affected by the commit: AbstractKAPSChangePropagationAnalysis, ArchitectureModelLookup, and BusChanges.
4. Evaluation Data

4.1.4. SmartGrid

For the model-based analysis SmartGrid, we identified ten historical evolution scenarios. The reproduction data for SmartGrid contains the classes of the ten scenarios. Each scenario is divided into two folders, the classes of the monolithic version is contained in the folder before and the classes of the modular version after the refactoring is contained in the after folder. Table 4.4 contains links to the monolithic and modular model-based analysis.

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>SmartGridLang [10]</td>
<td>–</td>
</tr>
<tr>
<td>Modular Language</td>
<td>mSmartGridLang [10]</td>
<td>–</td>
</tr>
<tr>
<td>Analysis</td>
<td>SmartGrid</td>
<td>SmartGrid GitHub [7]</td>
</tr>
<tr>
<td>Modular Analysis</td>
<td>mSmartGrid</td>
<td>mSmartGrid [7]</td>
</tr>
</tbody>
</table>

Table 4.4.: Overview SmartGrid Projects

4.1.4.1. Scenario 01 – Pass Data to Power Load Sim Properly

The scenario is the commit dfe199815. The commit occurred on Friday, November 24th 2017. In the monolith, one file is changed. See fig. A.41 and fig. A.42. The following file is affected by the commit: ReactiveSimulationController.

4.1.4.2. Scenario 02 – Added Report Generation for Attacker Simulation

The scenario is the commit c8280939. The commit occurred on Sunday, April 23th 2017. In the monolith, one file is changed. See fig. A.43 and fig. A.44. The following file is affected by the commit: ReportGenerator.

4.1.4.3. Scenario 03 – Fixed to Support String IDs

The scenario is the commit 72ecaa73. The commit occurred on Tuesday, October 17th 2017. In the monolith, two files are changed. See fig. A.45 and fig. A.46. The following files are affected by the commit: GraphAnalyzer, and Tarjan.

4.1.4.4. Scenario 04 – Added Init Methods with Maps as Parameter

The scenario is the commit 2d7a9c46. The commit occurred on Friday, February 7th 2020. In the monolith, eight files are changed. See fig. A.47 and fig. A.48. The following files are affected by the commit: LocalHacker, ViralHacker, HashMapHelper, GraphAnalyzer, IAttackerSimulation, IImpactAnalysis, ImpactAnalysisMock, and NoAttackerSimulation.

---

4.1. Evolution Scenarios

4.1.4.5. Scenario 05 – Added rootNode Search Viral Hacker

The scenario is the commit 1648636e. The commit occurred on Friday, November 22nd 2019. In the monolith, five files are changed. See fig. A.49 and fig. A.50 The following files are affected by the commit: LocalHacker, ViralHacker, ScenarioModelHelper, ReactiveSimulationController, and TestClientRMI.

4.1.4.6. Scenario 06 – Finalizing the RCP Commands

The scenario is the commit aae4a894. The commit occurred on Monday, July 27th 2020. In the monolith, eleven files are changed. See fig. A.51 and fig. A.52 The following files are affected by the commit: FileSystemHelper, Activator, SmartgridRCPApplication, ControllerCommand, GetModifiedPowerspecsCommand, InitTopoCommand, SimControlCommands, EObjectsHelper, LocalController, ReactiveSimulationController, and RCPCall.

4.1.4.7. Scenario 07 – Local Controller Without a Launch Configuration

The scenario is the commit 63ae1f4. The commit occurred on Friday, February 7th 2020. In the monolith, four files are changed. See fig. A.53 and fig. A.54 The following files are affected by the commit: ITimeProgressor, NoOperationTimeProgressor, LocalController, and ReactiveSimulationController.

4.1.4.8. Scenario 08 – Nodes are Now Randomly Hacked When Using Full Meshed Hacking

The scenario is the commit 3d81da9e. The commit occurred on Friday, January 15th 2016. In the monolith, one file is changed. See fig. A.55 and fig. A.56 The following file is affected by the commit: ViralHacker.

4.1.4.9. Scenario 09 – Modified Attacker Simulation to Support Disabling Root Node for Virus

The scenario is the commit 5ee72f70. The commit occurred on Tuesday, December 15th 2015. In the monolith, two files are changed. See fig. A.57 and fig. A.58 The following files are affected by the commit: LocalHacker, and ViralHacker.

4.1.4.10. Scenario 10 – Added Boolean Method to Attacker Simulation that Indicates if Attributes Can be Used or Not

The scenario is the commit 4c257bea. The commit occurred on Friday, November 13th 2015. In the monolith, two files are changed. See fig. A.59 and fig. A.60 The following files are affected by the commit: LocalHacker, and ViralHacker.
Bibliography


A. Appendix

A.1. Camunda
Figure A.1.: Camunda scenario 01 - before refactoring
A.1. Camunda

Figure A.2.: Camunda scenario 01 - after refactoring
Figure A.3.: Camunda scenario 02 - before refactoring
A.1. Camunda

Figure A.4.: Camunda scenario 02 - after refactoring

Figure A.5.: Camunda scenario 03 - before refactoring
Figure A.6.: Camunda scenario 03 - after refactoring
A.1. Camunda

Figure A.7.: Camunda scenario 04 - before refactoring
### A. Appendix

#### Figure A.8.: Camunda scenario 04 - after refactoring
A. Appendix

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ParadigmAbstractFlowNodeBuilder&lt;B, E&gt;</code></td>
<td>paradigms for building flow nodes</td>
</tr>
<tr>
<td><code>businessRuleTask()</code></td>
<td>Business Rule Task Builder</td>
</tr>
<tr>
<td><code>businessRuleTask(String)</code></td>
<td>Business Rule Task Builder</td>
</tr>
<tr>
<td><code>callActivity()</code></td>
<td>Call Activity Builder</td>
</tr>
<tr>
<td><code>callActivity(String)</code></td>
<td>Call Activity Builder</td>
</tr>
<tr>
<td><code>camundaAsyncAfter()</code></td>
<td>Camunda Async After</td>
</tr>
<tr>
<td><code>camundaAsyncAfter(boolean)</code></td>
<td>Camunda Async After</td>
</tr>
<tr>
<td><code>camundaAsyncBefore()</code></td>
<td>Camunda Async Before</td>
</tr>
<tr>
<td><code>camundaAsyncBefore(boolean)</code></td>
<td>Camunda Async Before</td>
</tr>
<tr>
<td><code>camundaExclusive()</code></td>
<td>Camunda Exclusive</td>
</tr>
<tr>
<td><code>camundaExclusive(boolean)</code></td>
<td>Camunda Exclusive</td>
</tr>
<tr>
<td><code>camundaExecutionListenerClass(String, Class)</code></td>
<td>Camunda Execution Listener Class</td>
</tr>
<tr>
<td><code>camundaExecutionListenerClass(String, String)</code></td>
<td>Camunda Execution Listener Class</td>
</tr>
<tr>
<td><code>camundaExecutionListenerDelegateExpression(String, String)</code></td>
<td>Camunda Execution Listener Delegate Expression</td>
</tr>
<tr>
<td><code>camundaFailedJobRetryTimeCycle(String)</code></td>
<td>Camunda Failed Job Retry Time Cycle</td>
</tr>
<tr>
<td><code>camundaJobPriority(String)</code></td>
<td>Camunda Job Priority</td>
</tr>
<tr>
<td><code>condition(String, String)</code></td>
<td>Condition</td>
</tr>
<tr>
<td><code>connectTarget(FlowNode)</code></td>
<td>Connect Target Flow Node</td>
</tr>
<tr>
<td><code>connectTargetWithAssociation(FlowNode)</code></td>
<td>Connect Target With Association Flow Node</td>
</tr>
<tr>
<td><code>connectTo(String)</code></td>
<td>Abstract Flow Node Builder</td>
</tr>
<tr>
<td><code>createTarget(Class&lt;?&gt;)</code></td>
<td>Create Target</td>
</tr>
<tr>
<td><code>createTarget(Class&lt;?&gt;, String)</code></td>
<td>Create Target String</td>
</tr>
<tr>
<td><code>createTargetBuilder(Class&lt;?&gt;, String)</code></td>
<td>Create Target Builder</td>
</tr>
<tr>
<td><code>createTargetBuilder(Class&lt;?&gt;, String)</code></td>
<td>Create Target Builder String</td>
</tr>
<tr>
<td><code>endEvent()</code></td>
<td>End Event Builder</td>
</tr>
<tr>
<td><code>eventBasedGateway()</code></td>
<td>Event Based Gateway Builder</td>
</tr>
<tr>
<td><code>exclusiveGateway()</code></td>
<td>Exclusive Gateway Builder</td>
</tr>
<tr>
<td><code>exclusiveGateway(String)</code></td>
<td>Exclusive Gateway Builder String</td>
</tr>
<tr>
<td><code>findLastGateway()</code></td>
<td>Gateway</td>
</tr>
<tr>
<td><code>inclusiveGateway()</code></td>
<td>Inclusive Gateway Builder</td>
</tr>
<tr>
<td><code>inclusiveGateway(String)</code></td>
<td>Inclusive Gateway Builder String</td>
</tr>
<tr>
<td><code>moveToActivity(String)</code></td>
<td>Move To Activity</td>
</tr>
<tr>
<td><code>moveToNextGateway()</code></td>
<td>Abstract Gateway Builder</td>
</tr>
<tr>
<td><code>moveToNode(String)</code></td>
<td>Abstract Flow Node Builder</td>
</tr>
<tr>
<td><code>notCamundaExclusive()</code></td>
<td>Not Camunda Exclusive</td>
</tr>
<tr>
<td><code>parallelGateway()</code></td>
<td>Parallel Gateway Builder</td>
</tr>
<tr>
<td><code>parallelGateway(String)</code></td>
<td>Parallel Gateway Builder String</td>
</tr>
<tr>
<td><code>receiveTask()</code></td>
<td>Receive Task Builder</td>
</tr>
<tr>
<td><code>receiveTask(String)</code></td>
<td>Receive Task Builder String</td>
</tr>
<tr>
<td><code>scriptTask()</code></td>
<td>Script Task Builder</td>
</tr>
<tr>
<td><code>scriptTask(String)</code></td>
<td>Script Task Builder String</td>
</tr>
<tr>
<td><code>sendTask()</code></td>
<td>Send Task Builder</td>
</tr>
<tr>
<td><code>sendTask(String)</code></td>
<td>Send Task Builder String</td>
</tr>
<tr>
<td><code>sequenceFlowBuilder(String)</code></td>
<td>Sequence Flow Builder</td>
</tr>
<tr>
<td><code>serviceTask()</code></td>
<td>Service Task Builder</td>
</tr>
<tr>
<td><code>serviceTask(String)</code></td>
<td>Service Task Builder String</td>
</tr>
<tr>
<td><code>subProcess(String)</code></td>
<td>Sub Process Builder</td>
</tr>
<tr>
<td><code>subProcess(String)</code></td>
<td>Sub Process Builder String</td>
</tr>
<tr>
<td><code>transaction()</code></td>
<td>Transaction Builder</td>
</tr>
<tr>
<td><code>transaction(String)</code></td>
<td>Transaction Builder String</td>
</tr>
<tr>
<td><code>boundaryEventWithStartedCompensation()</code></td>
<td>Boundary Event With Started Compensation</td>
</tr>
<tr>
<td><code>boundaryEventWithStartedCompensation()</code></td>
<td>Boundary Event With Started Compensation String</td>
</tr>
<tr>
<td><code>compensationHandler()</code></td>
<td>Compensation Handler</td>
</tr>
<tr>
<td><code>compensationHandler()</code></td>
<td>Compensation Handler String</td>
</tr>
</tbody>
</table>

Figure A.10.: Camunda scenario 05 - after refactoring
Figure A.11.: Camunda scenario 06 - before refactoring
Figure A.12.: Camunda scenario 06 - after refactoring
A.1. Camunda

Figure A.13.: Camunda scenario 07 - before refactoring
Figure A.14: Camunda scenario 07 - after refactoring
A.1. Camunda

Figure A.15.: Camunda scenario 08 - before refactoring
Figure A.16: Camunda scenario 08 - after refactoring
Figure A.17.: Camunda scenario 09 - before refactoring
A. Appendix

![Diagram](image)

Figure A.18.: Camunda scenario 09 - after refactoring
Figure A.19.: Camunda scenario 10 - before refactoring
Figure A.20.: Camunda scenario 10 - after refactoring
A.2. KAMP4aPS
Figure A.21: KAMP scenario 01 - before refactoring
Figure A.22.: KAMP scenario 01 - after refactoring

Figure A.23.: KAMP scenario 02 - before refactoring
Figure A.24.: KAMP scenario 02 - after refactoring
Figure A.25.: KAMP scenario 03 - before refactoring
Figure A.26.: KAMP scenario 03 - after refactoring

Figure A.27.: KAMP scenario 04 - before refactoring
Figure A.28.: KAMP scenario 04 - after refactoring

Figure A.29.: KAMP scenario 05 - before refactoring
Figure A.30.: KAMP scenario 05 - after refactoring

Figure A.31.: KAMP scenario 06 - before refactoring
Figure A.32.: KAMP scenario 06 - after refactoring

Figure A.33.: KAMP scenario 07 - before refactoring
Figure A.34.: KAMP scenario 07 - after refactoring
Figure A.35.: KAMP scenario 08 - before refactoring

Figure A.36.: KAMP scenario 08 - after refactoring
Figure A.37.: KAMP scenario 09 - before refactoring
Figure A.38.: KAMP scenario 09 - after refactoring
Figure A.39: KAMP scenario 10 - before refactoring
Figure A.40.: KAMP scenario 10 - after refactoring
A. Appendix

A.3. SmartGrid

Figure A.41.: SmartGrid scenario 01 - before refactoring

Figure A.42.: SmartGrid scenario 01 - after refactoring
A.3. SmartGrid

Figure A.43.: SmartGrid scenario 02 - before refactoring

Figure A.44.: SmartGrid scenario 02 - after refactoring

Figure A.45.: SmartGrid scenario 03 - before refactoring
Figure A.46.: SmartGrid scenario 03 - after refactoring

Figure A.47.: SmartGrid scenario 04 - before refactoring
A.3. SmartGrid

Figure A.48.: SmartGrid scenario 04 - after refactoring

Figure A.49.: SmartGrid scenario 05 - before refactoring
Figure A.50.: SmartGrid scenario 05 - after refactoring
Figure A.51.: SmartGrid scenario 06 - before refactoring
Figure A.52.: SmartGrid scenario 06 - after refactoring
Figure A.53.: SmartGrid scenario 07 - before refactoring
A. Appendix

Figure A.54.: SmartGrid scenario 07 - after refactoring

Figure A.55.: SmartGrid scenario 08 - before refactoring
A.3. SmartGrid

Figure A.56.: SmartGrid scenario 08 - after refactoring

Figure A.57.: SmartGrid scenario 09 - before refactoring
A. Appendix

Figure A.58.: SmartGrid scenario 09 - after refactoring

Figure A.59.: SmartGrid scenario 10 - before refactoring

Figure A.60.: SmartGrid scenario 10 - after refactoring