



# Beagle

Design and Architecture

Annika Berger, Joshua Gleitze, Roman Langrehr,
Christoph Michelbach, Ansgar Spiegler, Michael Vogt

10th of January 2016

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

| | |
|---|---|
| Reviewer: | Jun.-Prof. Dr.-Ing. Anne Koziolek |
| Advisor: | M.Sc. Axel Busch |
| Second advisor: | M.Sc. Michael Langhammer |

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

# Contents

# List of Figures

# Abbreviations

**API** Application Programming Interface

**CTA** Common Trace API

**GUI** Graphical User Interface

**PCM** Palladio Component Model

**RDIA** Resource Demanding Internal Action

**SEFF** service effect specification

**SRS** Software Requirements Specification

# 1 Architectural Overview

This chapter gives an introduction to Beagle's high level design. The following chapters will describe conceptual details of different subsystems. For the specification of single types, please refer to Beagle's Javdoc documentation [Berger et al., 2016].

Beagle consists of a core component and interfaces to external components. Components may depend on information provided by another component, but their internal logic works strictly independently. Communication takes exclusively place through the core component. The following are Beagle's key components and interfaces:

### Core Component (Mediator Pattern)

In order to manage and synchronise the requests and execution of different jobs, Beagle is controlled by a core component. The core component conducts the order of executable services, distributes information and is responsible for class instantiation. It contains all management logic required to perform dynamic analysis on software and will offer a parametrised Palladio Component Model (PCM) at the end of a successful execution. It does, however, not contain any logic to actually run measurements or analyse parametric dependencies. Instead, it depends on other components providing this functionality.

### Measurement Tool

Measurement Tools are responsible for all kinds of measurements that are needed to get the execution time of Resource Demanding Internal Actions(RDIAs), branch decisions of service effect specification (SEFF) Branches and repetitions of SEFF Loops in regard to a certain parametrisation. An adapter instructing Kieker will be the first class to implement this interface.

### Measurement Result Analyser

Based on the measurement results, Measurement Result Analysers will suggest Evaluable Expressions that describe the parametric dependencies found in the results. Typical implementations are regression tools.

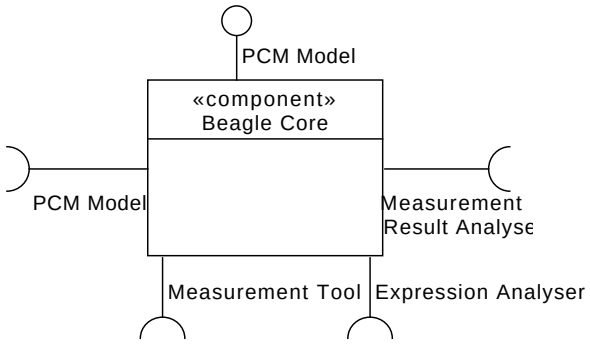


Figure 1.1: The Beagle Core Component and its interfaces.

### Proposed Expression Analyser

Proposed Expression Analysers try to improve the results of Measurement Result Analysers. They usually try to combine proposed expressions to build a better one (in terms of Beagle's Fitness Function). As different regression approaches usually have different advantages and shortcomings, combining their results may produce a more accurate expression because it contains "more parts of the truth" [Krogmann, 2011]. The genetic approach described by Krogmann would be implemented as a Proposed Expression Analyser.

### Final Judge

This class is responsible to decide which proposed evaluable expression describes the measured results best and will be annotated in the PCM. It also decides if more measurements should be done and when the final solution is found.

## 1.1 Extension Points

Measurement Tools, Measurement Expression Analysers and Proposed Expression Analysers are connected to Beagle via Eclipse Extension Points. This allows a flexible configuration of the used plugins after compilation. Development of the plugins can take place independently (because the Application Programming Interface (API) is fixed) and which plugins users install can depend on the software they have available.

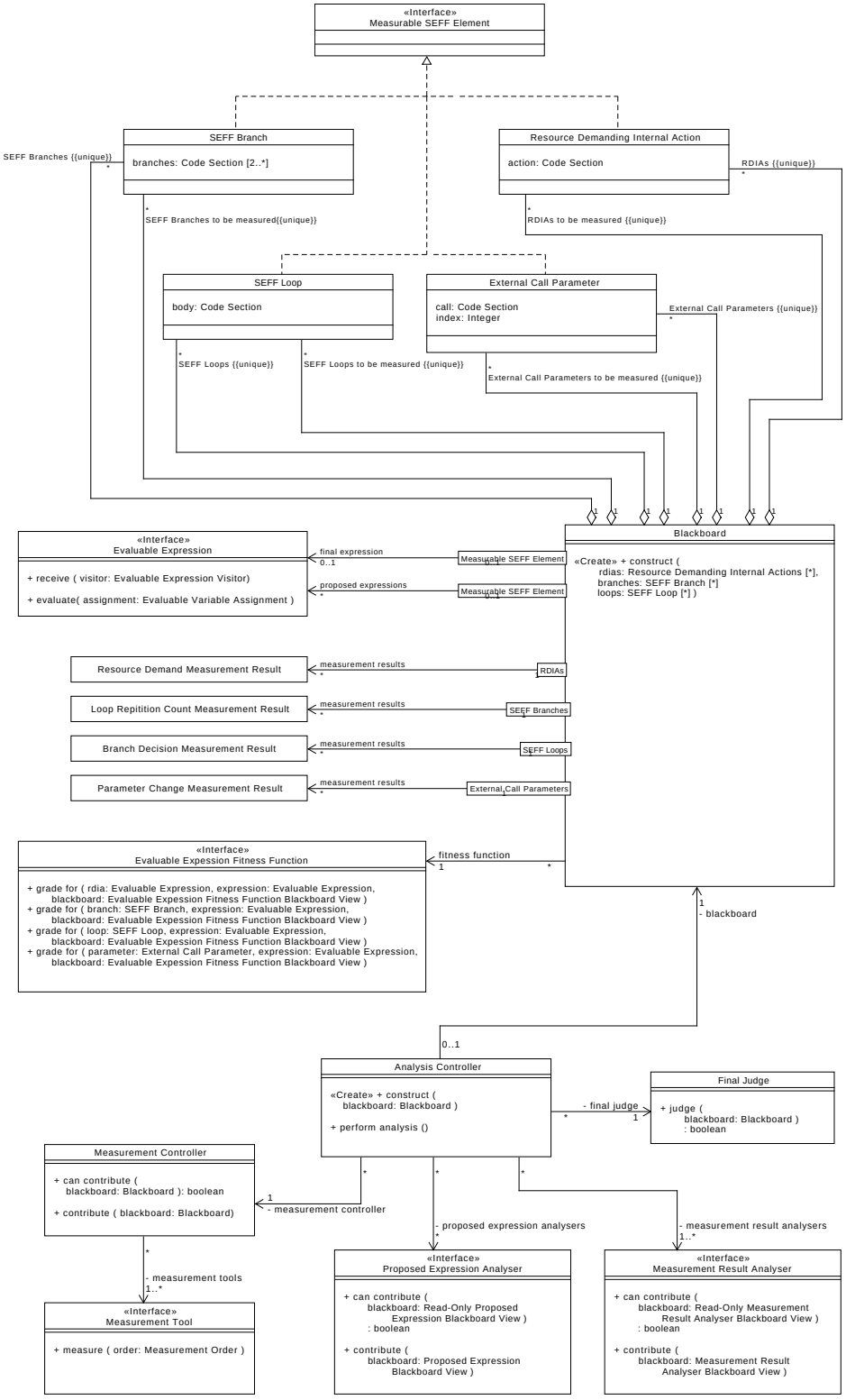


Figure 1.2: Class overview of Beagle Core. For details about specific classes, refer to Beagle's Javadoc [Berger et al., 2016].

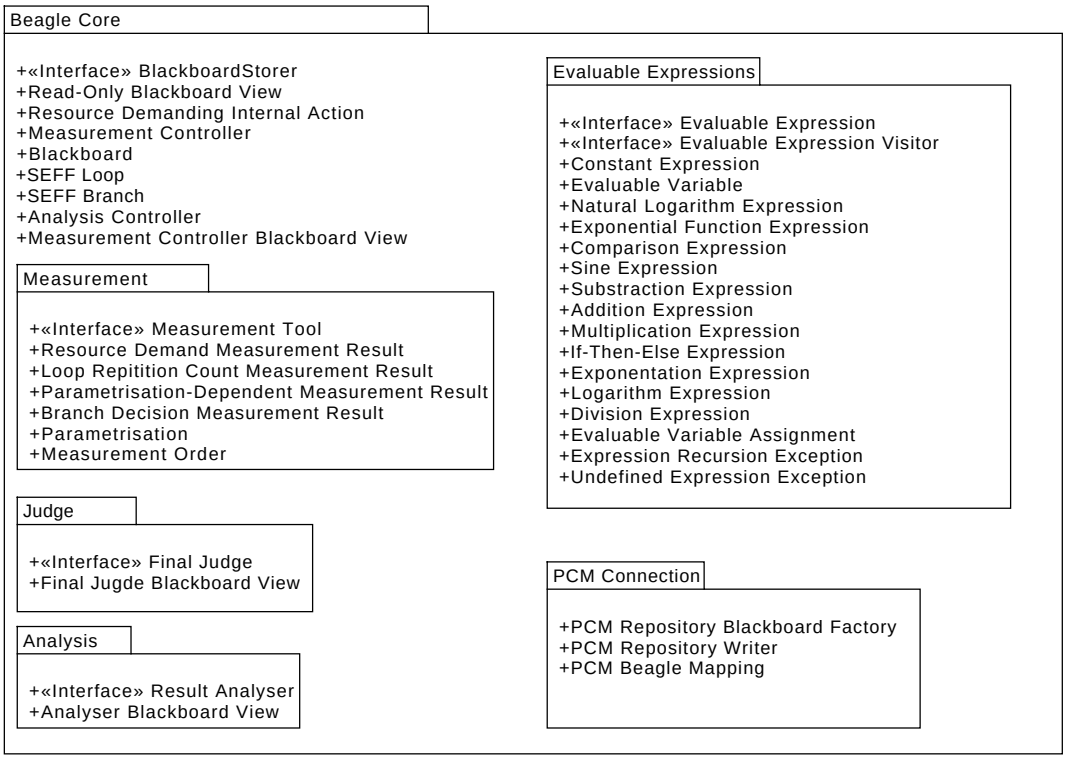


Figure 1.3: Beagle Core's separation into Packages.

# 2 Beagle’s Knowledge: The Blackboard

## 2.1 Measurable SEFF Elements

## 2.2 Evaluable Expressions

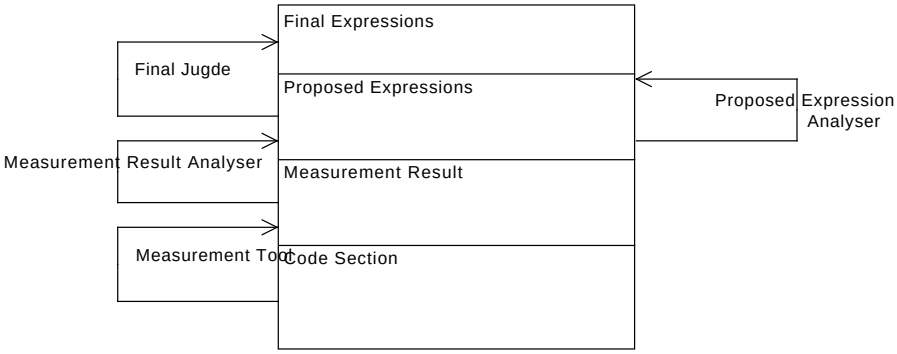


Figure 2.1: Abstraction Layers on the Blackboard

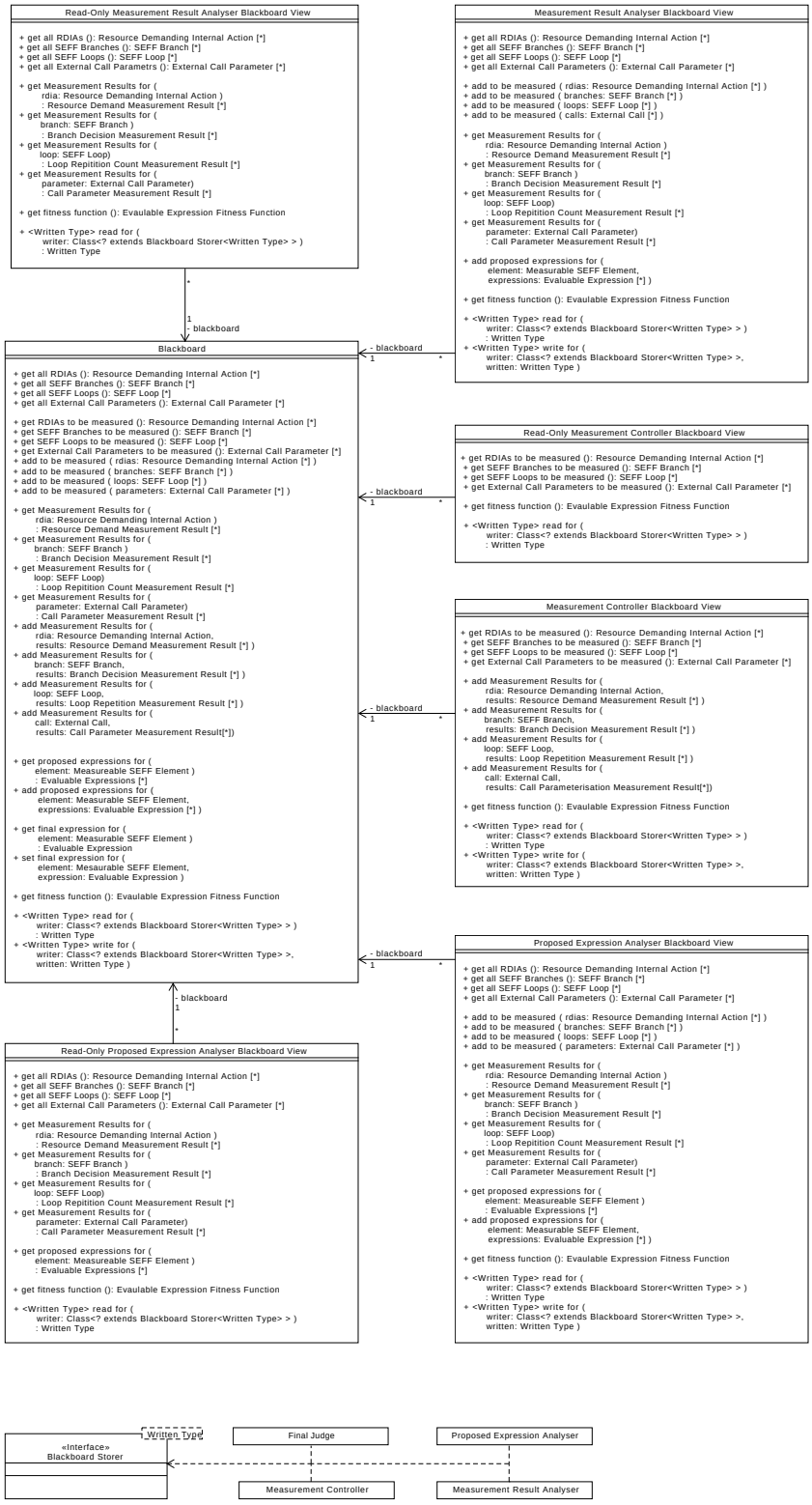


Figure 2.2: The Blackboard and its views. The accessor methods are explicitly stated to visualise the view's usage.

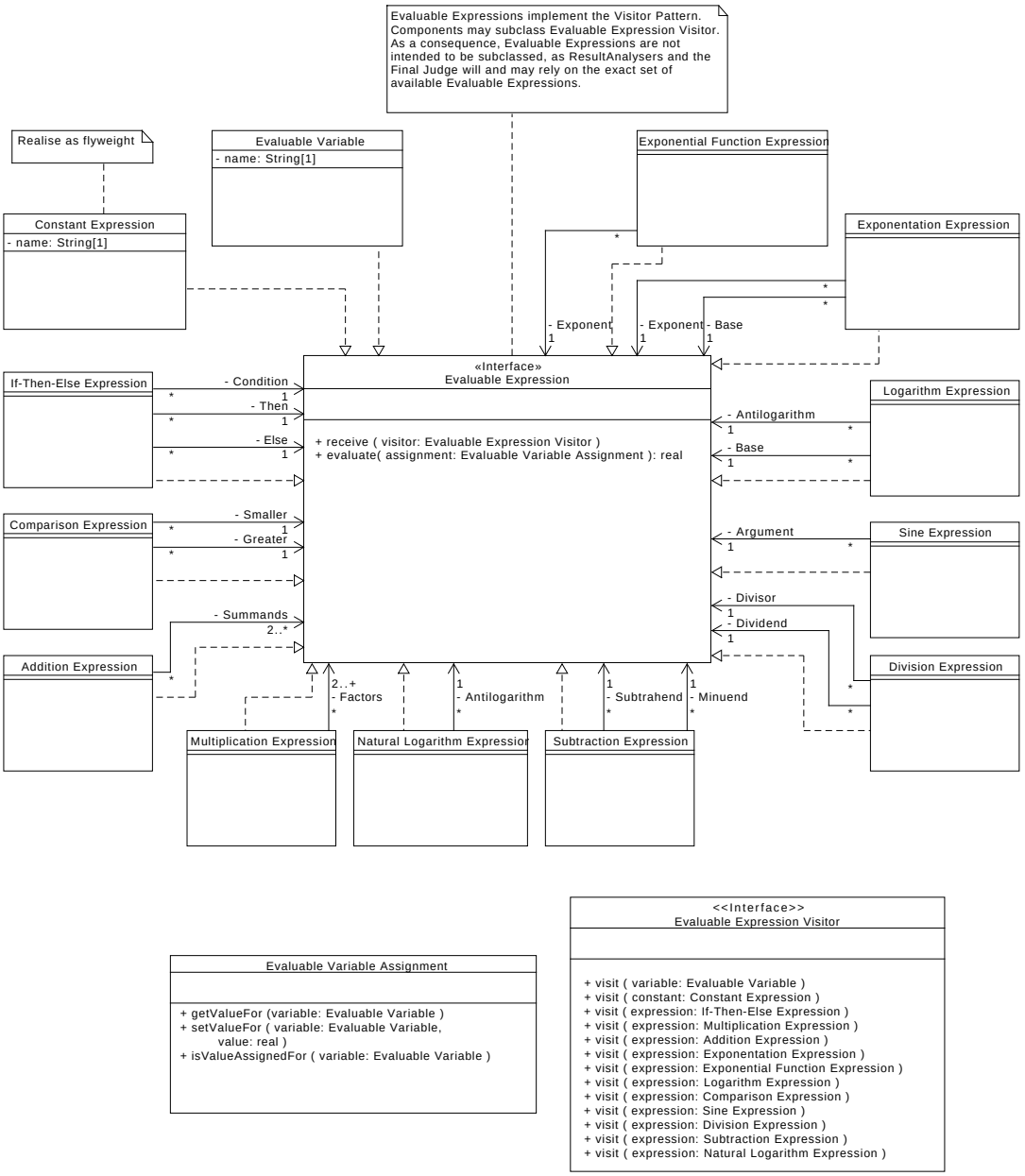


Figure 2.3: The Evaluable Expression Interface and its implementations realise the Visitor pattern.

# 3 The Analysis

At Beagle's core, the Analysis Controller controls all analysis activity by instructing the Measurement Controller, the Measurement Result Analysers, the Proposed Expression Analysers and the Final Judge. While not contributing itself, it is charge of all control flow during analysis.

Analysis Controller#perform Analysis performs a complete analysis, starting by measuring the examined software, and continuing to analyse until the Final Judge reports that the analysis is finished. There is always at most one Measurement Tool, Measurement Result Analyser, Proposed Expression Analyser or Final Judge having the control flow at any given moment during the execution of Analysis Controller#perform Analysis ("the analysis loop").

An iteration of the analysis loop starts by asking the Measurement Controller whether it wants to conduct measurements for the current blackboard state—which will usually be the case if there is something not yet measured—, and if so, calling its #measure method. The Measurement Controller will then instruct the Measurement Tools to measure. Usually, it will tell every tool to measure all new SEFF Elements.

After that, the main loop invokes one arbitrary chosen Measurement Result Analyser reporting to be able to contribute to the current blackboard state. This analyser may then propose expressions describing the parametric dependencies of SEFF Elements' measurement results. If there is no such analyser, an arbitrary chosen Proposed Expression Analyser reporting to be able to contribute will be invoked. It may then propose more expressions based on the ones the ones Measurement Result Analysers added to the blackboard, usually trying to improve them. If there Final Judge will be called. It decides whether enough information has been collected and Beagle can terminate. If this is the case, it also creates or selects the final result for each item that has proposed results.

The analysis loop will then be repeated until the Final Judge was called and its #judge method returned true. Figure 3.1 sketches the procedure.

## 3.1 Measurement

## 3.2 Final Judge

```
finished := false

romraBlackboardView := Read-Only Measurement Result Analyser Blackboard
View.construct(blackboard)
mraBlackboardView := MeasurementController Blackboard View.construct(blackboard)
ropeBlackboardView := Read-Only Proposed Expression Analyser Blackboard
View.construct(blackboard)
peBlackboardView := Proposed Expressions Blackboard View.construct(blackboard)

while ¬finished do
    if measurement controller.can measure(blackboard) then
        measurement controller.measure(blackboard)
    else if ∃ analyser ∈ measuremet result analysers :
    analyser.can contribute(romraBlackboardView) then
        analyser.contribute(mraBlackboardView)
    else if ∃ analyser ∈ proposed expression analysers :
    analyser.can contribute(ropeBlackboardView) then
        analyser.contribute(peBlackboardView)
    else
        finished := final judge.judge(blackboard)
    end
end
```

Figure 3.1: Beagle Controller#perform Analysis() in pseudocode.

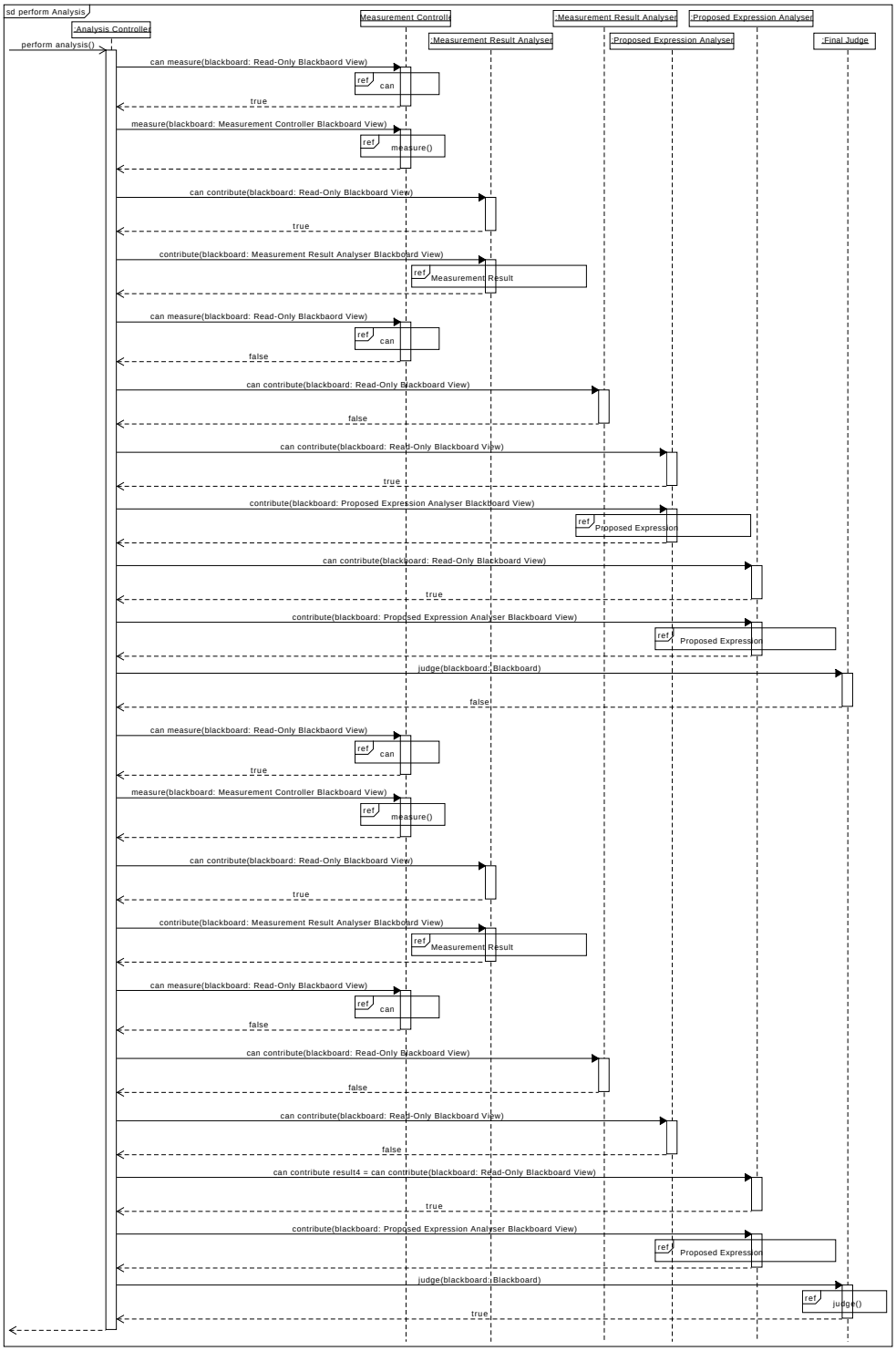


Figure 3.2: UML Sequence diagram for Analysis Controller#perform Analysis()

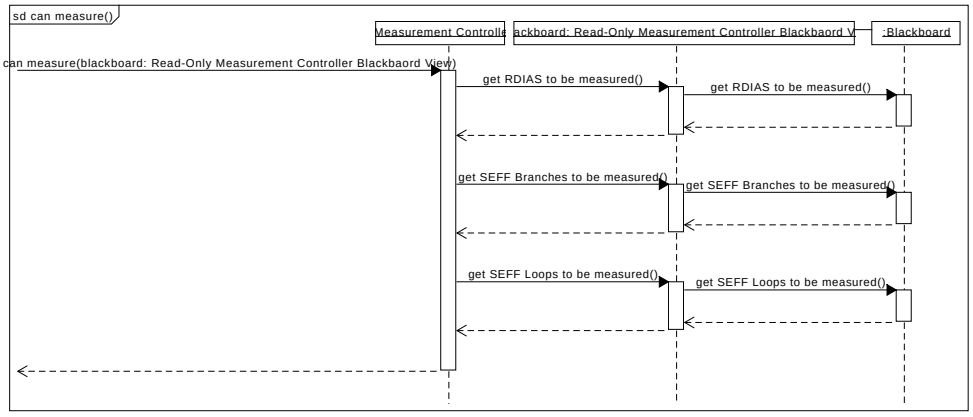


Figure 3.3: UML Sequence diagram for Measurement Controller#can measure()

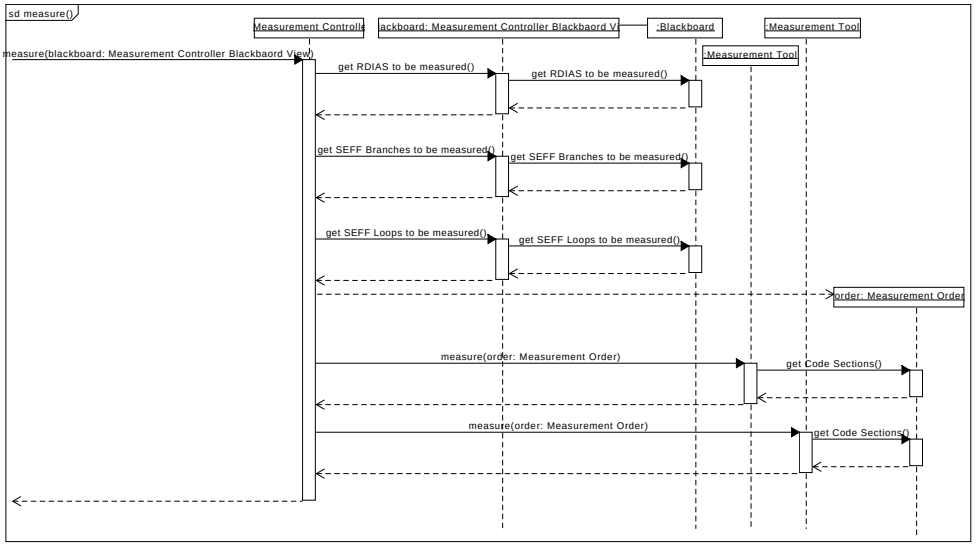


Figure 3.4: UML Sequence diagram for Measurement Controller#measure()

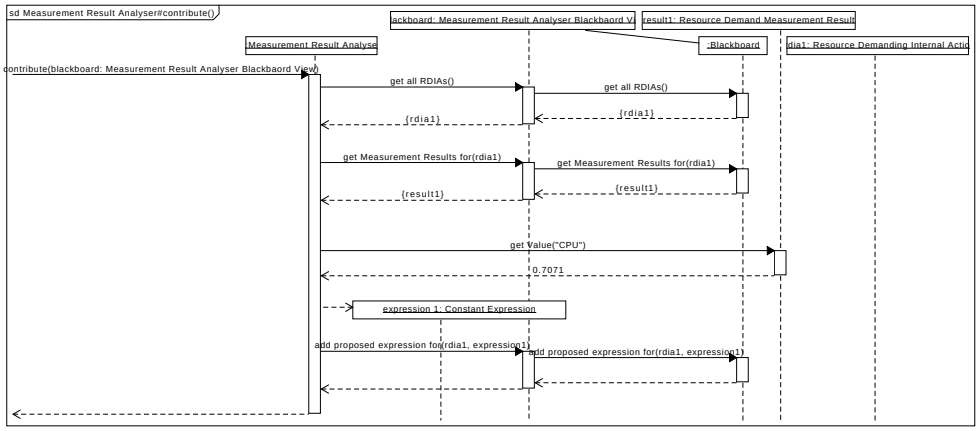


Figure 3.5: UML Sequence diagram for Measurement Result Analyser#contribute()

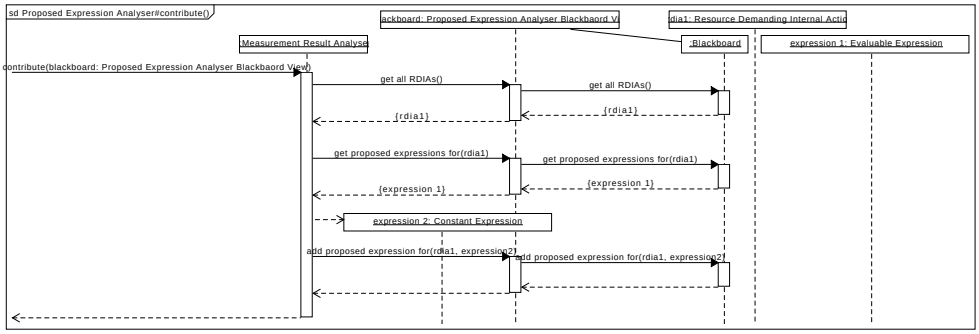


Figure 3.6: UML Sequence diagram for Proposed Expression Analyser#contribute()

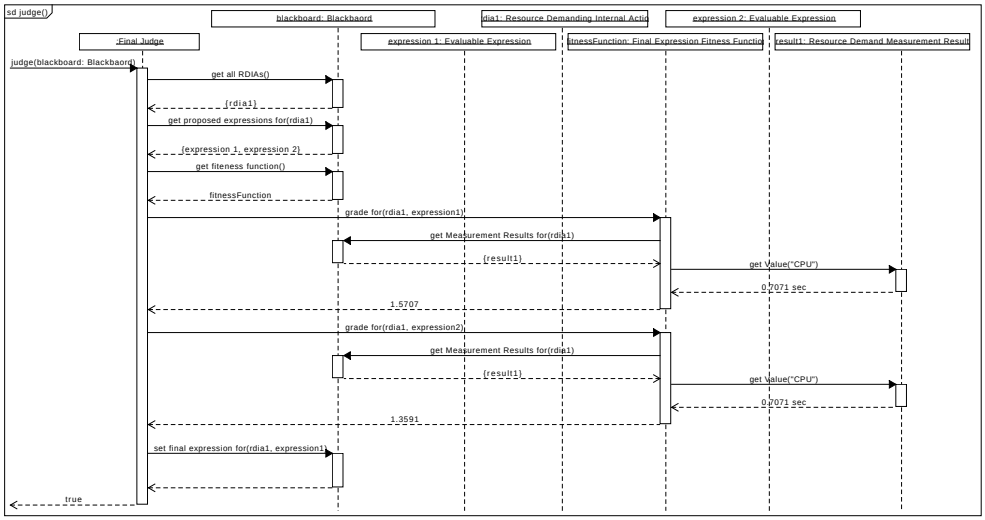


Figure 3.7: UML Sequence diagram for Final Judge#judge()

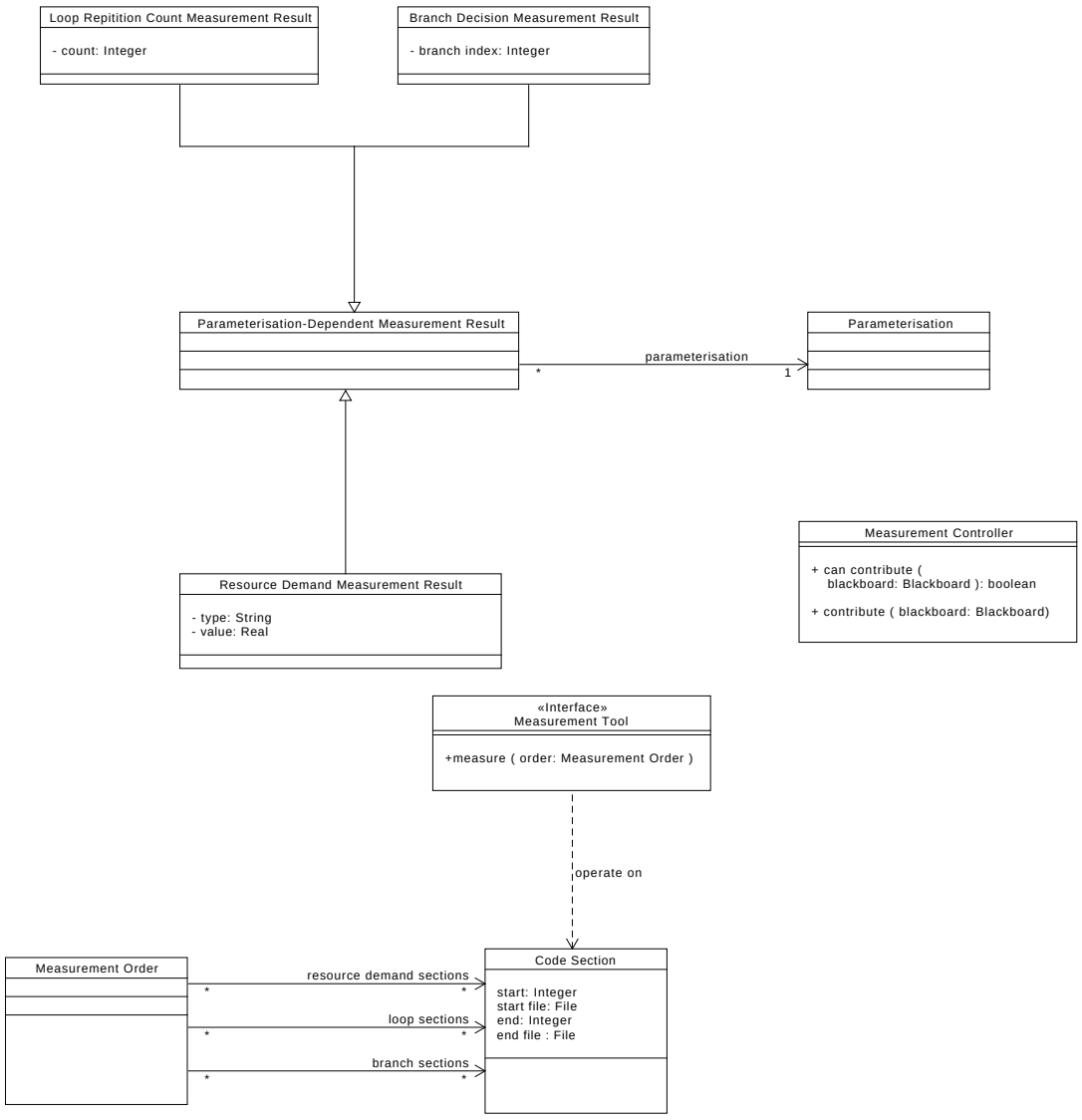


Figure 3.8: The Measurement Results.

# 4 Graphical User Interface

## 4.1 Design

The Graphical User Interface (GUI) follows the Model-View-Controller paradigm. It accordingly clearly separates responsibilities: Displaying GUI parts, controlling these parts, setting up the model (Beagle Core), and the model itself. This leads to a clear flow of information and control. Information is stored either locally or in a single object every GUI class knows (the User Configuration) because only the user's settings need to be transported through all objects. The user always has control over the analysis because information can be transported from the dialog, though the GUI Controller object, to the Beagle Controller which can instruct the model.

## 4.2 Control Flow

GUI Controller is the heart of Beagle's GUI. It is responsible to start the GUI. The class controls the flow of actions on the GUI and gives control to Beagle Controller once the user started the analysis. Beagle Controller is commanded by GUI Controller to start, pause, continue, and abort the analysis.

The GUI Controller is created and called from actions originating from context menu clicks. The GUI stores all data obtained from the user in the User Configuration. Thus, the GUI Controller creates a User Configuration which it from now on will be associated with. It then creates a new Beagle Analysis Wizard which in return will create its Wizard Pages. All objects obtain the User Configuration to be able to directly provide their gathered information to it. Calling GUI Controller's open() method will display the GUI and show the Analysis Wizard, prompting the use for all information necessary to perform an analysis.

After the wizard has finished, control is returned to the GUI Controller which will now (concurrently) instruct the Beagle Controller to set up the Beagle Core Component. To do so, Beagle Controller sets up a Blackboard based on the information collected in the User Configuration. For example, it uses a PCM Repository Blackboard Factory to translate PCM objects to the affected Measurable SEFF Elements. The prepared Blackboard is then passed to Beagle Core's Analysis Controller, which will perform the actual analysis.

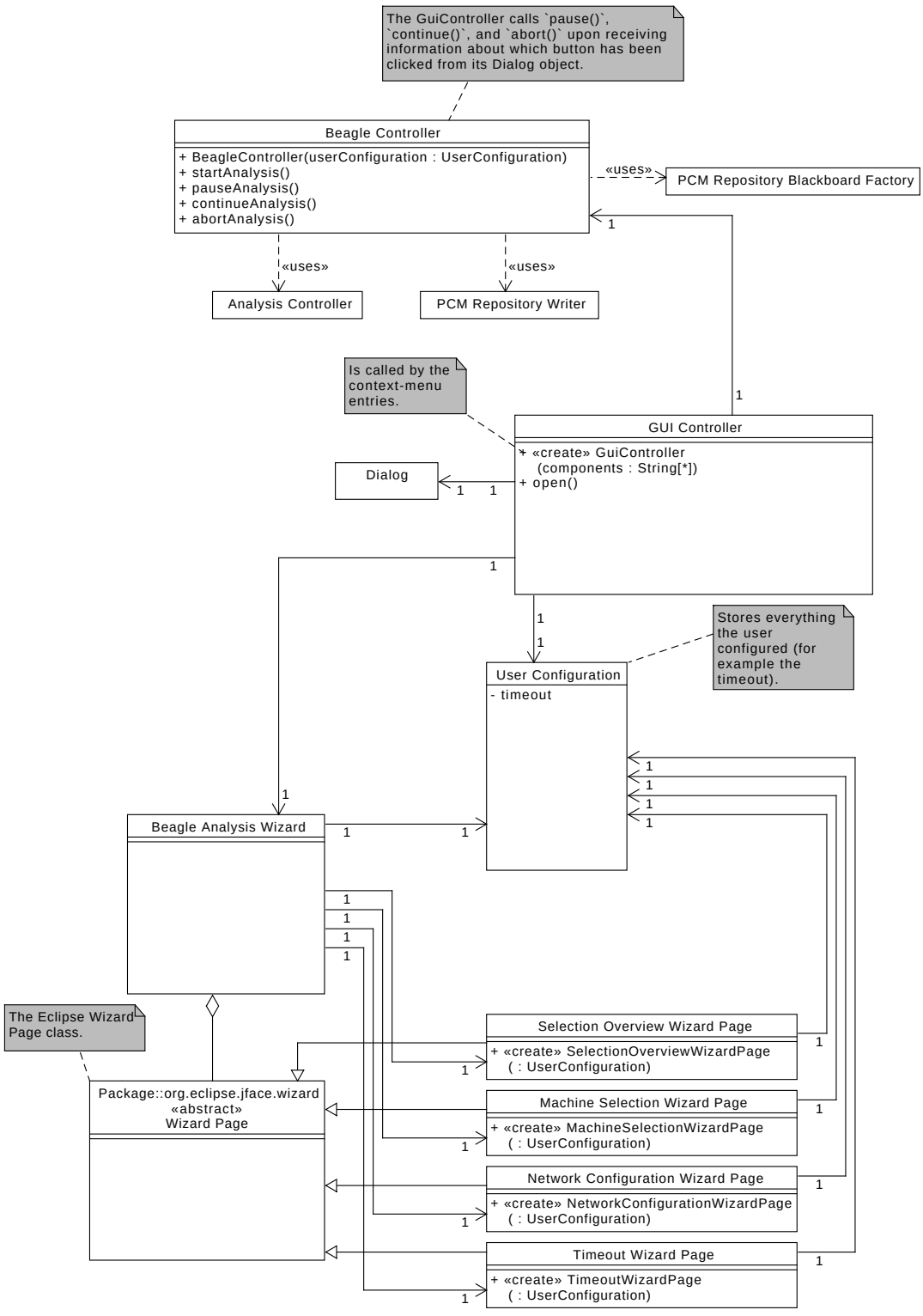


Figure 4.1: UML class diagram of the GUI classes.

In the meantime, GUI Controller opens a dialog providing the user with information about the analysis and options to pause, continue or abort the analysis. GUI Controller bequeaths these calls the corresponding methods of Beagle Controller, which handles the Beagle Core.

# 5 Requirements Specification

Beagle's software design directly follows the Software Requirements Specification (SRS) [Berger et al., 2015]. Most design decisions are proposed to fulfil mandatory requirements while allowing easily supplementing optional requirements. Section 5.1 describes changes to the SRS that proved necessary while designing Beagle. Section 5.2 describes how Beagle's requirements are reflected in its design. However, some mandatory criteria have changed, concerning the Common Trace API (CTA) .

## 5.1 Changes to the Software Requirements Specification

### The Common Trace API

The CTA was planned to be used by Beagle to communicate with measurement software (/B10/, /F40/). While starting to investigate the API, it became apparent that it does not offer the expected functionality. The CTA is designed to return measurement results on method level, while Beagle's measurements need to be performed on sub-method (statement) level. Furthermore, there is no possibility to instrument source code, or control measurement software, the CTA can only return measurement results. Beagle's Measurement Tools will now be directly connected to specific measurement software like Kieker, without the CTA as intermediary.

Accordingly, the following modifications where made to the Software Requirements Specification (SRS):

/B10/ Results are not transferred through the CTA.

/F40/ Has been removed without substitution.

/T30/ Has been removed without substitution.

/T210/ The CTA will not be tested.

### Measurement Timeout

The measurement timeout functionality described in /F50/ and /F60/ is a valuable function to Beagle. However, it is not required to successfully use Beagle for the purpose described in the Chapter 1 of the SRS. Denoting it as mandatory was a mistake.

Hence, the following modifications where made to the SRS:

/F50/ was moved to /OF70/

/F60/ was moved to /OF80/

### Layout

Further minor changes have been made to the SRS, fixing broken links and layout issues.

## 5.2 Relating Requirements to the Design

# Terms and Definitions

**Common Trace API**

an API developed by NovaTec GmbH for measuring the time, specific code sections need to be executed.

**internal action**

sequence of commands a component executes without leaving its scope (e.g. without calling other components). Part of a component's SEFF.

**Kieker**

> "a Java-based application performance monitoring and dynamic software analysis framework." [van Hoorn et al., 2012]

A measurement software Beagle aims to support.

**Palladio Component Model**

a domain-specific modelling language (DSL) used by Palladio.

> It is designed to enable early performance predictions for software architectures and is aligned with a component-based software development process. [Kounev, 2009]

**resource demand**

how much of a certain resource—like Central Processing Unit (CPU), Network or hard disk drive—a component needs to offer a certain functionality. In the PCM, resource demands are part of the SEFF. They are ideally specified platform independently, e.g. by specifying required CPU cycles, megabytes to be read, etc. If such information is not available, resource demands can be expressed platform dependent, e.g. in nanoseconds. In this case, a certain degree of portability can still be achieved if information about the used platforms' speed relative to each other is available.

**service effect specification**

description of a component's behaviour in the PCM. SEFFs contain information about the component's calls to other components as well as its resource demands. This information is used to derive the component's performance for simulation and prediction.

# Bibliography

[Berger et al., 2015] Berger, A., Gleitze, J., Langrehr, R., Michelbach, C., Spiegler, A., and Vogt, M. (2015). Beagle—software requirements specification. Technical report, Karlsruhe Intitute of Technology Department of Informatics Institute for Program Structures and Data Organization (IPD).

[Berger et al., 2016] Berger, A., Gleitze, J., Langrehr, R., Michelbach, C., Spiegler, A., and Vogt, M. (2016). Beagle—javadoc. Technical report, Karlsruhe Intitute of Technology Department of Informatics Institute for Program Structures and Data Organization (IPD).

[Kounev, 2009] Kounev, S. (2009). *Automated extraction of palladio component models from running enterprise Java applications*. PhD thesis, University of Wuerzburg.

[Krogmann, 2011] Krogmann, K. (2011). *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*. PhD thesis, Karlsruhe Institute of Technology.

[van Hoorn et al., 2012] van Hoorn, A., Waller, J., and Hasselbring, W. (2012). Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM.