Beagle

Software Requirements Specification

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

29th of November 2015

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

Reference notation

This document uses a fixed notation for all of its contents, making them referenceable:

/P#/ purpose criterion

/B#/ purpose boundary

/A#/ application attribute

/G#/ target group

/E#/ software environment attribute

/D#/ data

/F#/ functional requirement

/Q#/ non functional requirement

/C#/ challenge or assumption

/T#/ test case

/M#/ model

A preceding “O” marks optional points. These relate to features that are desired and planned, but can not surely be implemented in the project’s scope. They also serve as an outlook for further development.

1 Purpose and Goals

When developing software, specifying its architecture in a sophisticated way is a crucial, yet challenging task. Decisions made at this point highly influence the software's quality of service (QoS), but are usually difficult to change, as redesigns may be costly [Reussner et al., 2011]. To prevent poor design in the first place, Palladio, a model driven approach for software simulation, enables developers to analyse component-based softwares' QoS at the definition phase, before actually writing any code. Using Palladio, all parties involved in the development of component-based software model their domain in the Palladio Component Model (PCM). This information is hence used to simulate the software's behaviour, with a focus on its QoS attributes.

In many scenarios however, some to all source code may already exist. Analysis with Palladio might still be wished, for example to simulate a component's interaction with a software system or to freshly start analysing existing software. For such cases, SoMoX, a software for static source code analysis, allows users to re-engineer their software's architecture into a PCM. The results contain the software's component boundaries, their bindings to the provided source code and their service effect specification (SEFF) [Krogmann, 2011]. Unfortunately, SoMoX' static approach does not allow it to determine the software's resource demands, which are essential for performance analysis.

[Krogmann, 2011] also describes Beagle, an approach for dynamic source code analysis to complement SoMoX. It aims to conduct performance measurements on source code of a software, in order to determine its component's internal actions resource demands. Adding this information to the software's PCM enables developers to import their software into Palladio with minimal effort. The purpose of this project is to implement Beagle. Based on the foundations in [Krogmann, 2011], it aims to develop such a software adding dynamic properties to a PCM using contemporary measurement software.

1.1 Criteria

Mandatory

/P10/ Beagle enables the user to analyse given source code regarding the resources its internal actions' demand when executed.

/P20/ Beagle annotates its resource demand findings in a given instance of the software's PCM, enabling the user to import existing software into Palladio for analysis.

Optional

/OP10/ Beagle analyses given source code for further dynamic behavioural attributes.

1.2 Boundary

/B10/ Beagle does not perform actual measurements on source code. This is done by other software like Kieker and results are transferred through the Common Trace API (CTA).

/B20/ Beagle does not reconstruct a model of software's architecture from its source code. This is done by other software like SoMoX.

/B30/ Beagle does not reconstruct the internal structure of components like their SEFF. This is done by other tools like SoMoX.

/B40/ Beagle does not assert that analysis of source code written in a language other than Java 6 is possible.

/B50/ Beagle does no performance analysis or prediction. This is may be achieved with Palladio.

2 Application

2.1 Application Field

/A10/ Beagle can be used to re-engineer source code. To start to using Palladio for an existing software, Beagle can be combined with a tool for static code analysis like SoMoX. This way, the software can quickly be analysed with Palladio. Modelling an existing software is such a time-consuming task that automatic modelling is a valuable feature that may be crucial for developers to start using Palladio.

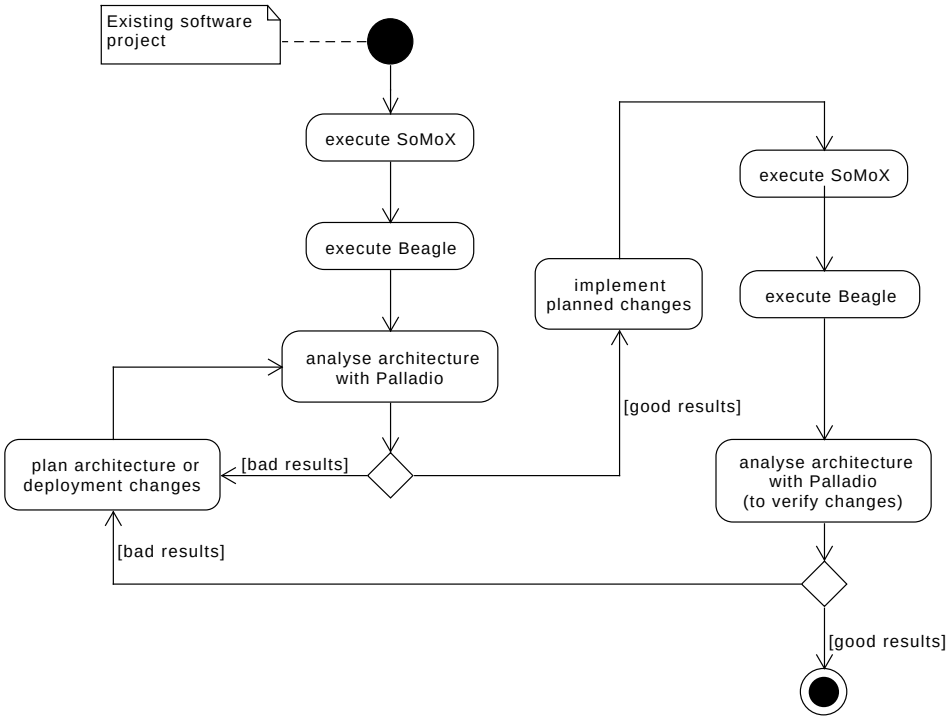

Figure 2.1: A typical workflow when using Beagle to re-engineer existing source code (/A10/).

/A20/ Beagle can be used for software development. Early implementations of components modelled in the PCM can be analysed with Beagle in order to predict their performance in interaction with the software system. This leads to earlier detection of arising problems (like implementation errors or unrealistic modelling in the PCM), which can hence be fixed in time.

/A30/ Beagle may be used for prototyping. Different implementations of a component modelled in the PCM may be analysed with Beagle to determine their resource demands. Palladio can then be used to simulate the software system's performance with each implementation. As performance is multi-dimensional, this can lead to more precise information about the different implementation's effects on the system's runtime.

/A40/ Beagle can be used to verify software's design and implementation. After developing the software with Palladio and implementing it, a static code analysis tool like SoMoX and Beagle can be re-engineer a PCM which can then be compared to the initial one. With this approach, differences and problems in the implementation can be detected and resolved easier.

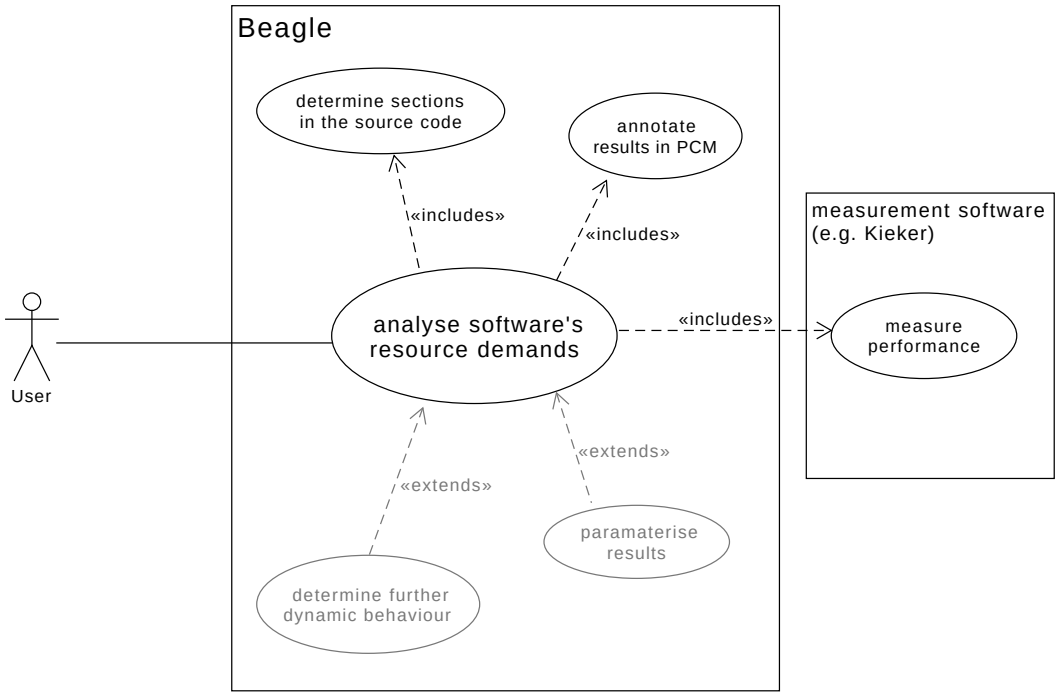

Figure 2.2: Use Case Diagram. Optional features are drawn grey.

2.2 Target Group

/G10/ Software architects will use Beagle predominantly for /A10/ and /A40/.

/G20/ System deployers will use Beagle predominantly for /A40/.

/G30/ Component developers will use Beagle predominantly for /A20/ and /A30/.

3 Environment

/E10/ Beagle should run on a Java 8 runtime environment (or higher) and Eclipse distribution that is up to date with Eclipse Mars (4.5).

/E20/ Beagle requires a PCM instance modelling the software to be analysed. The model must contain all components and their SEFFs, the source code and the decorater model.

/E30/ Beagle requires the CTA to communicate with performance measurement software.

/E40/ The user should not run other programms while Beagle is running as this disturbs the measurement. To receive optimal measurements Beagle should run on a dedicated server.

3.1 Component Model

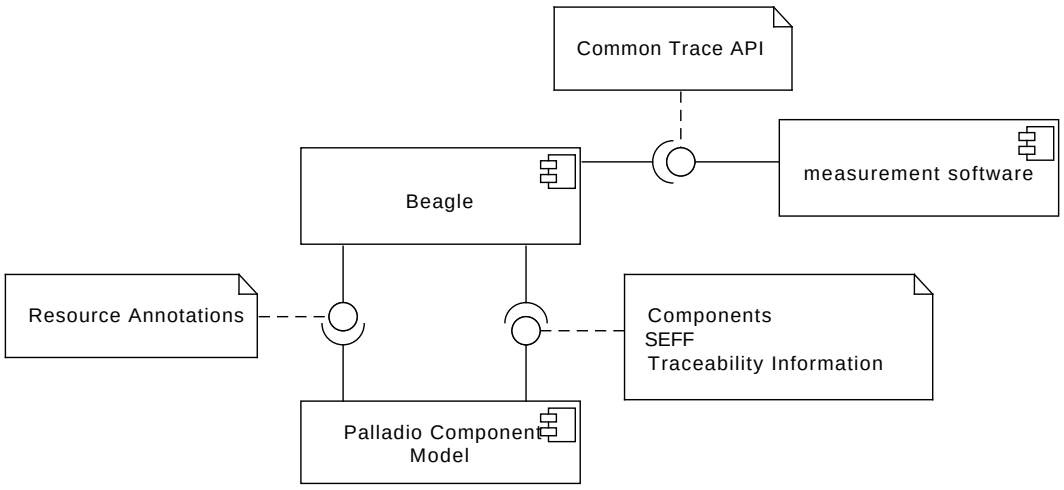

Figure 3.1: Beagle and its interaction with other software

4 Data

In the following chapter, “the software” refers to the software a user wants to analyse with Beagle. The term refers not only to source code, but also its conceptional attributes, like its purpose, structure and architecture.

Beagle deals with two major data artefacts: The software’s source code and an instance of the PCM describing the software (hereafter to be called “input source code”, “input PCM” or simply “input artefacts ”). Beagle will use the provided data to execute its tasks and write its results back into the PCM instance (hereafter to be called “result PCM ”) afterwards.

4.1 Input

Mandatory

/D10/ The software’s source code. It must be written in Java and either

- be provided together with `.class` files compiled out of it, such that the files are executable on a Java Runtime Environment (JRE) installed on the computer Beagle runs on, or
- be compilable by a JDK installed on the computer Beagle runs on.

/D20/ Information about the software’s components. They must be modelled in the input PCM.

/D30/ Information about the software’s components’ SEFFs. They must be modelled in the input PCM.

/D40/ Mappings of the software’s components to the parts in the source code implementing them.

Optional

/OD10/ User provided information about the software’s parts he wishes to analyse.

/OD20/ User provided information about measurement timeouts. May be provided prior to or during Beagle’s execution.

4.2 Output

Mandatory

/D100/ The software's components' internal actions' execution time.

Optional

/OD100/ The software's components' internal actions' further resource demands, like hard disk or network usage.

/OD110/ Probabilities of branches to be taken SEFF conditions.

/OD120/ Probable number of repeats in SEFF loops.

/OD130/ Measurement status data, containing all information required to resume a measurement (see /OF130/, /OF140/, enables /OF160/).

/OD140/ Verification data to check whether input artefacts changed (enables /OF150/).

5 Functional Requirements

Given Beagle is called with valid input artefacts (see p. 9), it must fulfil the following requirements:

5.1 Measurement

Mandatory

/F10/ Using the information provided in the PCM, Beagle determines the sections in the source code to be measured in order to find internal actions' resource demands.

/F20/ Beagle conducts measurements of the sections found by /F10/ by utilising measurement softwares.

/F30/ Beagle uses existing measurement software for /F20/.

/F40/ Beagle supports the CTA to communicate with measurement software.

/F50/ Beagle does not modify the provided source code files.

/F60/ Beagle stops measurements by an adaptive timeout when enabled. This means that it stops measurements if the software fails to respond within a certain period of time. This timeout is adapted based on previous runs, so that measurements of code sections that were fast in earlier measurements are aborted sooner if they fail to respond until then than measurements of sections known to be slow.

/F70/ The user can disable the adaptive timeout described in /F60/ and replace it with a set timeout or disable the timeout entirely.

Optional

/OF10/ Beagle approximately determines relations between components' interface parameters and their resource demands.

/OF20/ Beagle determines the probability for each case to be taken in encountered SEFF conditions.

/OF30/ Beagle determines /OF20/ depending on the component's interface parameters.

/OF40/ Beagle determines the probable number of repeats in encountered SEFF loops.

/OF50/ Beagle determines /OF40/ depending on the component's interface parameters.

5.2 Control

Optional

/OF100/ The user may choose whether Beagle will analyse the whole source code or only parts of it.

/OF110/ The user may choose to re-measure the source code or parts of it, in order to either gain more precision or to reflect source code changes.

/OF120/ The user may launch and control a measurement running on another computer over a network.

/OF130/ The user may pause and resume a measurement. Pausing causes all measurement activity to stop. Resuming continues the measurement from where they were paused.

/OF140/ The user may resume a paused measurement (/OF130/), even if Beagle was closed after pausing it.

/OF150/ Beagle asserts that no input artefact (p. input artefacts) was changed between pausing (/OF130/) and resuming (/OF140/) a measurement to assure its result's integrity.

/OF160/ Beagle's results do not change, no matter how often the user chooses to pause and resume the measurements.

/OF170/ The user may specify the parametrisation of a component.

/OF180/ If requested by the user, Beagle shuts down the computer it's running on after it finished a measurement.

5.3 Result Annotation

Mandatory

/F200/ Beagles stores all its results in the software's PCM ("result PCM", see p. 9).

/F210/ The result PCM is a valid PCM instance.

/F220/ As far as technically possible, Beagle's results can be read from the result PCM by a Palladio installation without Beagle.

/F230/ The result PCM contains all contained components' internal actions' resource demands.

/F240/ Any information found in the PCM instance provided to Beagle will be found in the result PCM.

/F250/ Measurement results are saved onto a persistent medium to avoid data loss

Optional

/OF200/ If Beagle found parametrised results (e.g. in /OF10/, /OF30/, /OF50/), they are expressed using the PCM Stochastic Expression Language.

6 Non Functional Requirements

6.1 Dependencies

Mandatory

/Q10/ In order to use Beagle, the user is not required to have any software installed but Java, Eclipse, Palladio and a measurement software supported by Beagle.

/Q20/ Beagle does not depend on any specific measurement software.

/Q30/ Beagle does not require its input artefacts to be generated by a specific software.

Optional

/OQ10/ Beagle can be used on every combination of operating system and hardware platform Eclipse and Palladio runs on.

/OQ20/ No user interaction is required while Beagle conducts measurements.

/OQ30/ Beagle shall handle any error thrown by the measured software, like uncaught Exceptions or calls to `System#exit`. This means that such errors do not influence other measurements.

/OQ40/ Beagle runs benchmarks on hardware systems in order to provide information to make its results portable.

6.2 User Interface and Experience

Mandatory

/Q100/ Beagle is implemented as an Eclipse plugin. As both Palladio and its extensions are Eclipse plugins, this ensures good usability for users.

/Q110/ Beagle can be controlled by context sensitive menus in Eclipse.

Optional

/OQ100/ Beagle is integrated into SoMoX to automatically be executed after SoMoX has finished.

/OQ110/ Beagle can obtain its input artefacts from SoMoX, such that the user does not need to provide further information after SoMoX was started. If Beagle requires more information than SoMoX provides, the user can already submit it while configuring SoMoX.

/OQ120/ Beagle reports its progress to the user.

6.2.1 GUI model

The user has multiple options to launch the analysis:

1. To analyse the whole project, there is an entry “Analyse with Beagle” in the context menu of the `.repository` or `.repository_diagram` file in eclipse.
2. To analyse a single component, there is an entry “Analyse with Beagle” in the context menu of each component in the repository diagram.
3. To analyse a single internal action, there is an entry “Analyse with Beagle” in the context menu of each internal action in the SEFF diagram.

   If an analysis with Beagle is not possible in option 1, 2 or 3, the context menu entry will be shown greyed and a description why the analysis is not possible appears, when the users tries to start the analysis.
4. If /OQ100/ is realised, the user has the option, when launching SoMoX to automatically launch the analysis with Beagle after SoMoX has finished.

When the users launches the analysis, he gets an window where he can change some settings for Beagle:

1. If /OQ40/ is realised, the user may adapt the default timeout.
2. If /OF120/ is realised, the connection to the measurement machine can be set up.
3. If /OQ40/ is realised, the user can select to additionally benchmark its hardware system.

If /OQ40/ is realised, Beagle also provides a button for benchmarking the hardware without running any analysis.

When the analysis is running, a window reporting progress is displayed.

In this window there is also button for pausing, if /OF130/ is realised. If the measurement is paused, this button changes to a resume button. If /OF140/ is implemented and the user chooses to close Eclipse, a dialogue allowing him to resume the measurement will appear every time he starts Eclipse afterwards. This dialogue also offers the options to disable the dialogue in the future and to abort the measurement and drop the data collected so far. Additionally, in each context menu having an entry “Analyse with Beagle”, there will be another entry called “Resume last Beagle measurement”, allowing the user to resume the measurement.

In the progress window, there is also a button to abort the measurement.

7 Test Cases

As Beagle has to work with the above defined interfaces it has to be tested in complete. However, this could result into testing the other software, which is not what should be done, or even worse it could result in not detecting errors and failures as they are compensated by other software. On account of this two types of tests are needed: one testing the whole system with its dependencies and another with parameters put in at the interfaces and therefore only testing Beagle itself.

Mandatory

/T10/ Assert that software is starting, running and terminating by a simple run-through. For a valid input this has to work without exceptions and the software has to terminate.

/T20/ Assert that Beagle discovers all sections needed for measurement and that they are correct. For a correct result in the PCM it is essential to to measure at the correct points. A part of this can be realised by checking if all code sections (of the measured part) were measured. Tests /F10/.

/T30/ Assert that Beagle works for a system with only the software specified in /Q10/. A new system has to be set up with only these software applications and Beagle has to be tested on it.

/T40/ Assert that transferring of data between interfaces and Beagle works correctly in boh ways. All interfaces with other software have to be tested.

/T50/ Assert that in PCM all measured resource demands are added and nothing else is changed. This includes to assure that the PCM is valid. Tests /C10/, /C20/, /F200/, /F210/, /F230/ and /F240/.

/T60/ Assert that Beagle measures the sections with measurement software through the CTA. Tests /F20/, /F30/ and /F40/.

/T70/ Assert that Beagle stops measurements after timeout. Therefore run a measurment for software (which e.g. does not terminate) and define an timeout. Additional assert that this timeout can be turned off. Tests /F60/ and /F70/.

/T80/ Assert that results are stored in the PCM by a manual test: run a measurement and check the PCM afterwards. Tests /F200/.

/T90/ Assert that the result PCM can be read by a Palladio installation without Beagle by opening it on such a system. Tests /F220/.

/T100/ Assert that all the result PCM contains all gained information (see /F230/) by checking if it is similar to a before defined PCM. Test /F230/ and /F240/.

Optional

/OT10/ Assert that Beagle works for different operating systems and hardware by running measurements on different systems. Tests /OQ10/.

/OT20/ Assert that Beagle detects invalid input (e.g. if source code decorator does not fit to the code) and does not crash but responds to it in a acceptable way.

/OT30/ Assert that Beagle works with different software through the CTA. Therefore it is necessary to test Beagle with Kieker and other measurement software. Tests /F40/, /Q20/ and /OQ30/.

/OT40/ Assert that Beagle does not change source code files and other input except from the PCM. Tests /F50/.

/OT50/ Tests /OF10/, /OF20/ and /OF40/.

/OT60/ Assert that Beagle reacts to different component's interface parameters. Tests /OF30/ and /OF50/.

/OT70/ Assert that it is possible for users to decide whether the whole source code or only parts of it are analysed. Therefore do several runs and determine the different parts which have to be tested. Tests /OF100/.

/OT80/ Assert that users are able to re-measure source code by measuring same source code several times. Tests /OF110/.

/OT90/ Assert that pausing and resuming measurements works as defined in /OF130/, /OF140/, /OF150/ and /OF160/ by testing same source code without pausing and several different numbers of pauses. The result has to be the same as long as nothing else changes.

/OT100/ Assert that Beagle shuts down the computer if requested by a manual test. First, run a measurement and activate shutting down and then check if it worked and the results are saved. Tests /OF180/.

/OT110/ Assert that no user interaction is needed for measurements by running several measurements without user interaction. Tests /OQ20/.

/OT120/ Assert that it is possible to run Beagle automatically after SoMoX has finished by having a SoMoX run with activated automatic Beagle run. Another test run has to assert that between both runs no user interaction is needed. Tests /OQ100/ and /OQ110/.

8 Discussion

8.1 Assumptions

/C10/ The measured software was built using component-based software architecture. This assumption is derived from working with Palladio, which was built for analysing component-based software. Fortunately, it most of the time imposes little loss of generality, as any object oriented software can be described using terms of component-based software architecture (regarding each class as a component in the worst case). Such software will naturally not have the advantages that come with the component-based software approach, but might still be analysed for their performance.

/C20/ The measured software has a constant, deterministic runtime for a fixed configuration of input parameters, when ignoring influences of the hardware, operating system and error of measurement. This will be the case for most software. The fact the user tries to measure the software when using Beagle implies he expects it to behave in such a manner.

/C30/ The input artefacts (see p. 9) are integer. This means that all parts of the provided PCM describe the software correctly, completely and exactly like implemented in the source code. Beagle relies on this to be true and may produce inaccurate or wrong results if it is not.

This assumption will not cause problems if the PCM was re-engineered from the software's source code. But if the model and implementation diverged at any point (likely during the software's implementation), it may, however, lead to unexpected results.

8.2 Challenges

/C100/ There are a lot of factors influencing a CPU's performance: operating temperature, number of other processes, previous load, and data in cache, to name just a few. Beagle aims to find ways to compensate these factors. This may involve disabling Turbo Boost on Intel CPUs, reading the cores'

temperature and making sure the CPU is in a real world application thermal state, and further measures.

/C110/ Beagle must ensure the transferability and scalability of its measurement results across different hardware platforms. This stretches from software running on an average desktop pc via servers through to clusters of servers. Different hardware platforms vary in many different dimensions (CPU frequency, number of CPU cores, size and distribution of CPU caches, speed of RAM, network speed, hard disk throughput, etc.), yet the results have to be representative.

/C120/ Beagle measures only one component at a time, yet components interact with each other. If a component calls another component that takes a long time to return or does not return at all (e.g. because it is waiting for some input or it is defective), it might be wished or required to mock this component. Whether imitation of other components' behaviour can be done automatically needs investigation. Such imitations may not affect the measurement results.

/C130/ On modern operating systems, multitasking is the default. Users are used to work on multiple tasks at the same time and have multiple programs running. This could, however, influence Beagle's measurement results. If a considerable impact on measurement results is recognised, strategies to avoid them can be developed. These may including prompting the user to close certain applications.

/C140/ Beagle aims to parametrise its measurement results by the component's interface parameters. Such parametrisation will likely be described by a regression function $\mathbb{R}^n \to \mathbb{R}$. This elicits multiple challenges:

- regression of multi-dimensional functions is a challenging task.
- the regression functions might likely not be continuous.
- It is unclear what the real number representation of an arbitrary Java object might be.

Note that even if not all of the above points can be fully resolved, approximate parametrisation might still produce better results than no parametrisation at all.

9 Models

9.1 Scenario

Imagine, that a Java based online shop is running on a middle-class web server of a company named “EmmaSun”. During the first few years the software could deal with almost 99.9% of the requests and orders that are handled quite well without any delay. After an enormous expansion since the last year, the user numbers are currently growing for about 5 percent each week. Although the current servers are designed to fulfil a distinctly higher amount of user requests, the administration reported some few dropouts as well as increasing waiting times in single applications. Unfortunately, the software is based on an early design that has grown over years with missing documentation in many cases. The effort to re-write the complete software is an impossible act. The only solution is, to reanalyse the software’s source code and hopefully find any bottlenecks that can be repaired with lesser effort. But reanalysing source code is also a quite unmanageable task. So at this point, Beagle is used.

Beagle helps the team of software architects that was commissioned by EmmaSun to analyse the whole software. As Beagle depends on a PCM, the architects use the reversed-engineering plug-in SoMoX to create a valid PCM including all software components and their SEFFs. Providing Beagle a monitoring software named Kieker, that matches the Common Trace API, Beagle starts with conducting measurements on single software components, extending the PCM. After around 6 hours of measuring, the results of measuring are added into the PCM and Beagle calls Palladio to do a performance prediction. The prediction indicates an architectural violation of some software components, that lead to a a huge amount of sub-function calls through hierarchical layers.

The software architects decide to add an extra cache, that can store the results of the sub-function calls and make them available immediately. Fortunately, before they turn this idea into reality, they can adopt this design decision into the PCM. The little changes in the PCM lead to a much better performance prediction and the software architects agree to reimplement the new design.

Terms and Definitions

Common Trace API an API developed by NovaTec GmbH for measuring the time, specific code sections need to be executed. . 2, 25

component "a [software] unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition." [Szyperski, 2002] There is no equivalent of components in modern programming languages, in particular, a component usually consists of multiple Java classes. Components can be nested.. 1, 2, 4, 7, 9–13, 16, 23–25

component developer "[specifies] the functional and extra-functional properties of their components. They put the specification as well as the implementation in repositories, where software architects can retrieve them." [Heiko Koziolek, 2007] In the PCM, component developers create service effect specifications to define components' behavioural properties and store modelling and implementation artefacts in repositories. [Reussner et al., 2011] . 5

component-based software a software constituted of components.. 1, 23

component-based software architecture a software architecture utilising the concept of component-based software, therefore taking advantage of the reusability of its parts and preserving the same for newly created components. . 23

CTA Common Trace API. 2, 7, 11, 19, 20

internal action sequence of commands a component executes without leaving its scope (e.g. without calling other components). Part of a component's SEFF.. 1, 10, 11, 13, 16

Java Runtime Environment a software set containing a Java Virtual Machine, a browser plugin, the Java standard libraries, and a configuration tool. The Java Virtual Machine it contains is needed to run Java applications or applets. . 9

JRE Java Runtime Environment. 7, 9

Kieker "a Java-based application performance monitoring and dynamic software analysis framework." [van Hoorn et al., 2012] A measurement software Beagle aims to support. . 2, 20, 25

measurement software software capable of measuring the time, given source code needs to execute some task. The software's results are usually returned in a time unit like nanoseconds. Beagles interacts with such software through the CTA and uses it to find resource demands. . 1, 7, 11, 15, 20

Palladio an approach for the definition of component-based software architectures with a special focus on performance properties. . 1–4, 13, 15, 20, 23, 25

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.

[tuC, 2015] . 1

PCM Palladio Component Model. 1, 2, 4, 7, 9, 11–13, 19, 20, 23, 25

PCM Stochastic Expression Language expression language used by the PCM to define random variables. These variables can for example be used to specify glsplresource demand. Random variables can be defined using basic mathematic operations, common stochastic distributions and interface parameters [Reussner et al., 2011]. . 13

QoS quality of service. 1

quality of service a software's extra-functional attributes, like performance, reliability, maintainability or security. . 1

resource demand how much of a certain resource—like 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. . 1, 2, 4, 10, 11, 13, 19, 24

SEFF service effect specification. 1, 2, 7, 9, 16, 25

SEFF condition conditions (like Java's `if`, `if-else` and `switch-case` statements) which affect the calls a component makes to other components. Such conditions are—contrary to conditions that stay within an internal action—modelled in the component's SEFF.. 10, 11

SEFF loop loops (like Java's `for`, `while` and `do-while` statement) which affect the calls a component makes to other components. Such loops are—contrary to loops that stay within an internal action—modelled in the component's SEFF.. 10, 12

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. . 1

software architect developer role in the component-based software development process. Leads the development process by designing the software's architecture from existing or planned components and interfaces. Usually delegates the specification of required components to component developers. Uses architectural styles and patterns, analyses architectural specifications, and makes design decisions. In the PCM, software architects create the assembly model, specifying how existing components are composed.[Reussner et al., 2011] . 5, 25

software architecture the high-level structure and design of a software system as well as the discipline of creating and documenting these.. 1, 2, 9

SoMoX a Palladio plugin for static code analysis to re-engineer a software's architecture from its source code. Constructs a PCM instance including the reconstructed components and their SEFF.. 1–4, 16, 21, 25

system deployer developer role in the component-based software development process. Specifies the resource environment and allocates components to resources. Resources can both be hardware resources (CPU, hard disk, network connection) and software resources (thread pool, database connection). In the PCM, system deployers create the resource environment specification, modelling the resource environment and component allocations. [Reussner et al., 2011] . 5

List of Figures

Bibliography

[con,]

[tuC, 2015] (2015). Palladio component model.

[Heiko Koziolek, 2007] Heiko Koziolek, Jens Happe, S. B. R. R. (2007). *Palladio Paper*. PhD thesis.

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

[Reussner et al., 2011] Reussner, R., Becker, S., Burger, E., Happe, J., Hauck, M., Koziolek, A., Koziolek, H., Krogmann, K., and Kuperberg, M. (2011). The palladio component model. Technical report, Department of Informatics Institute for Program Structures and Data Organization (IPD).

[Szyperski, 2002] Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[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.