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

List of Figures iii

Abbreviations iv

Reference notation v

1 Purpose and Goals 1
- 1.1 Criteria . . . 2
- 1.2 Boundary . . . 2

2 Application 5
- 2.1 Application Field . . . 5
- 2.2 Target Group . . . 5

3 Environment 11
- 3.1 Component Model . . . 11

4 Data 13
- 4.1 Input . . . 13
- 4.2 Output . . . 14

5 Functional Requirements 15
- 5.1 Measurement . . . 15
- 5.2 Control . . . 16
- 5.3 Result Annotation . . . 17

6 Non-Functional Requirements 19
- 6.1 Dependencies . . . 19
- 6.2 User Interface and Experience . . . 19
  - 6.2.1 GUI Model . . . 20

7 Test Cases 23
- 7.1 Functionality . . . 23
- 7.2 Integration . . . 25

8 Discussion **27**
8.1 Assumptions 27
8.2 Challenges 28

9 Models **31**
9.1 Scenario 1 31
9.2 Scenario 2 32

Terms and Definitions **35**

Bibliography **41**

List of Figures

1.1 UML Use Case Diagram 3

2.1 UML Activity diagram for /A10/ 6
2.2 UML Activity diagram for /A20/ 7
2.3 UML Activity diagram for /A30/ 8
2.4 UML Activity diagram for /A40/ 9

3.1 Component Model 11

Abbreviations

CTA Common Trace API

GUI Graphical User Interface

JRE Java Runtime Environment

PCM Palladio Component Model

QoS quality of service

SEFF service effect specification

Reference notation

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

/A#/ application attribute

/B#/ purpose boundary

/C#/ challenge or assumption

/D#/ data

/E#/ software environment attribute

/F#/ functional requirement

/G#/ target group

/M#/ model

/P#/ purpose criterion

/Q#/ non functional requirement

/T#/ test case

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 during 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 then 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 desired: 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 software's source code 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 a piece of software adding dynamic properties to a PCM using contemporary measurement software.

It's important do understand the difference between simple performance measurements and the detection of resource demands. While the first one simply determines how long a piece of code need to execute a given task on a given machine, the latter respects the implications of component-based software architecture. This means Beagle tries to gain resource demand information that is independent from a component's usage,deployment, and assembly context.

1.1 Criteria

Mandatory

/P10/ Beagle enables users 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 users to import existing software into Palladio for analysis.

Optional

/OP10/ Beagle analyses consigned source code for further dynamic behavioural attributes like the number of loop executions in SEFF loops and the probability for branches in SEFF conditions.

/OP20/ Beagle outputs the results as a function of the input parameters of the internal actions as resource demand annotations in the PCM.

/OP30/ Beagle offers users a Graphical User Interface (GUI) to control the analysis .

1.2 Boundary

/B10/ Beagle does not perform actual measurements on source code. This is done by other software like Kieker. Their results are transferred to Beagle using 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 neither do performance analysis nor prediction. That is may be achieved with Palladio.

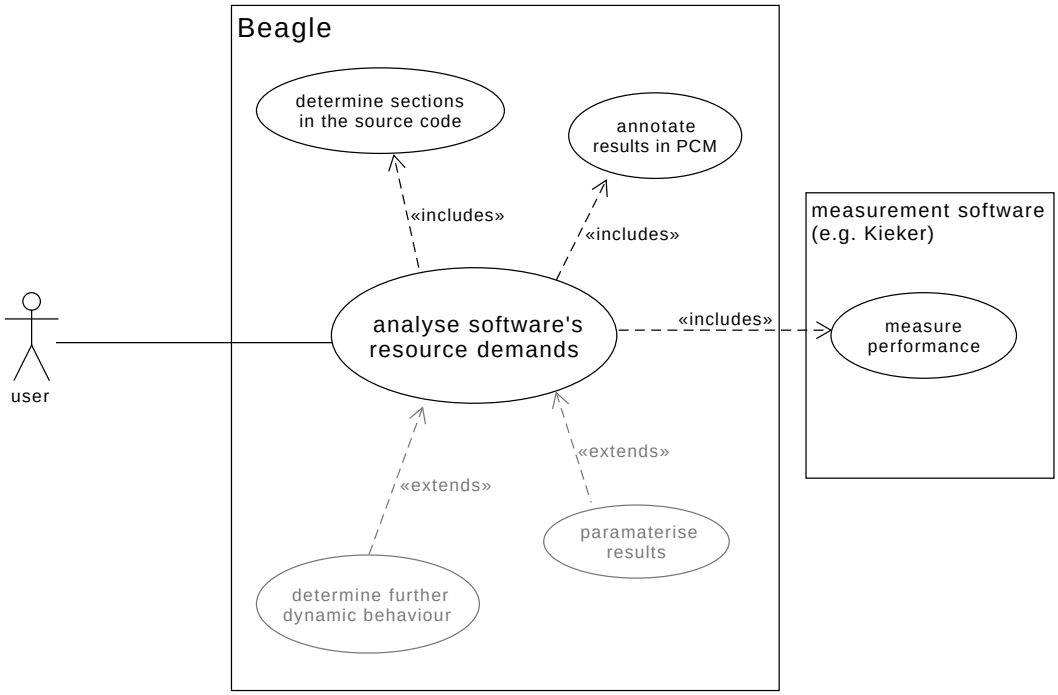

Figure 1.1: UML Use Case Diagram. Optional features are drawn grey.

2 Application

2.1 Application Field

/A10/ Beagle can be used to re-engineer source code. To start 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 which may be crucial to have for developers who start using Palladio.

/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 fast detection of arising problems (like implementation errors or unrealistic modelling in the PCM), which can then be fixed early on.

/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, Beagle can analyse its resource demands which can then be compared to the predicted ones. With this approach, differences and problems in the implementation can be detected and resolved more easily.

2.2 Target Group

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

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

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

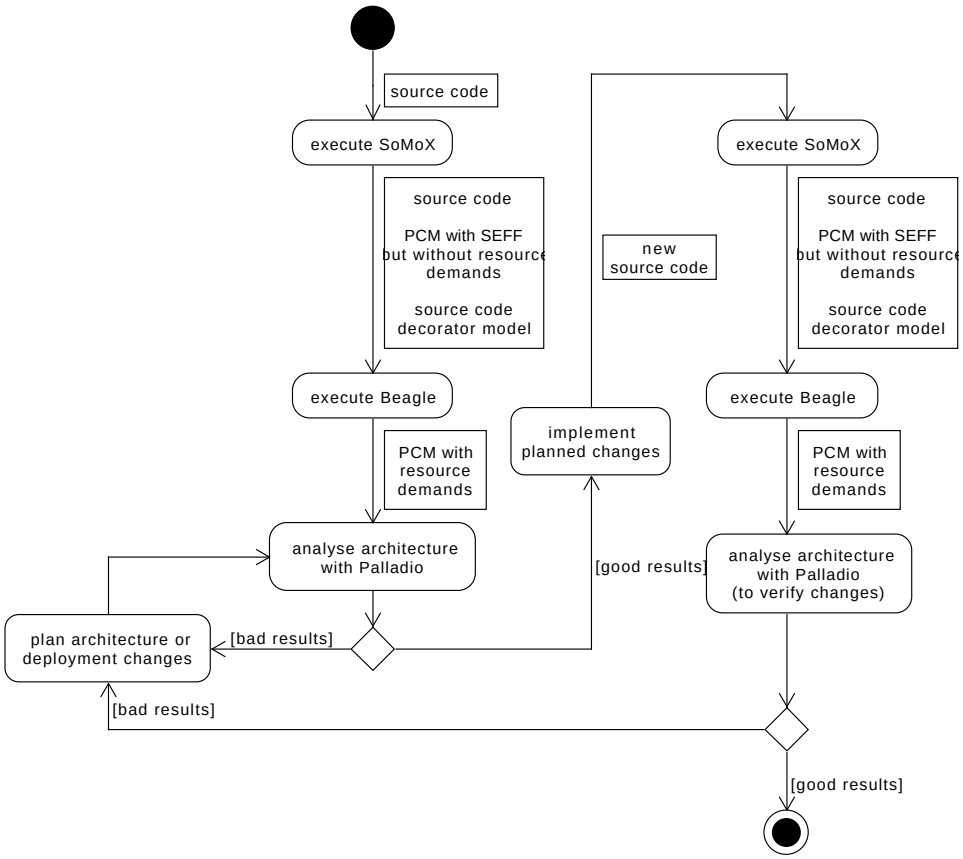

Figure 2.1: A typical workflow when using Beagle to re-engineer existing source code (/A10/).

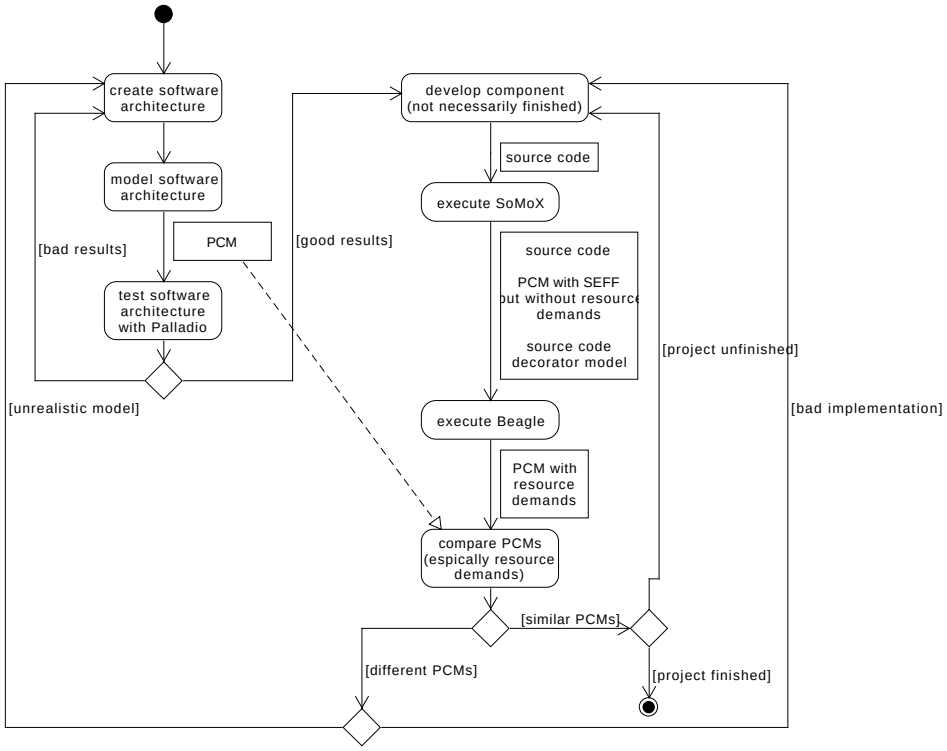

Figure 2.2: A typical workflow when using Beagle during software development (/A20/).

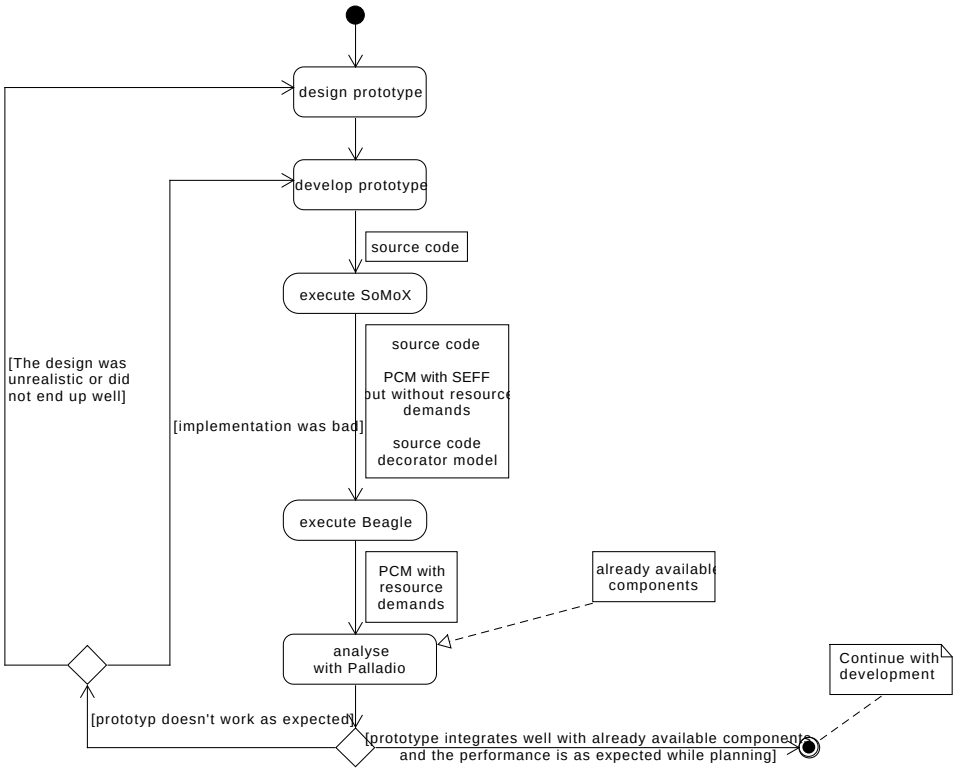

Figure 2.3: A typical workflow when using Beagle for prototyping (/A30/).

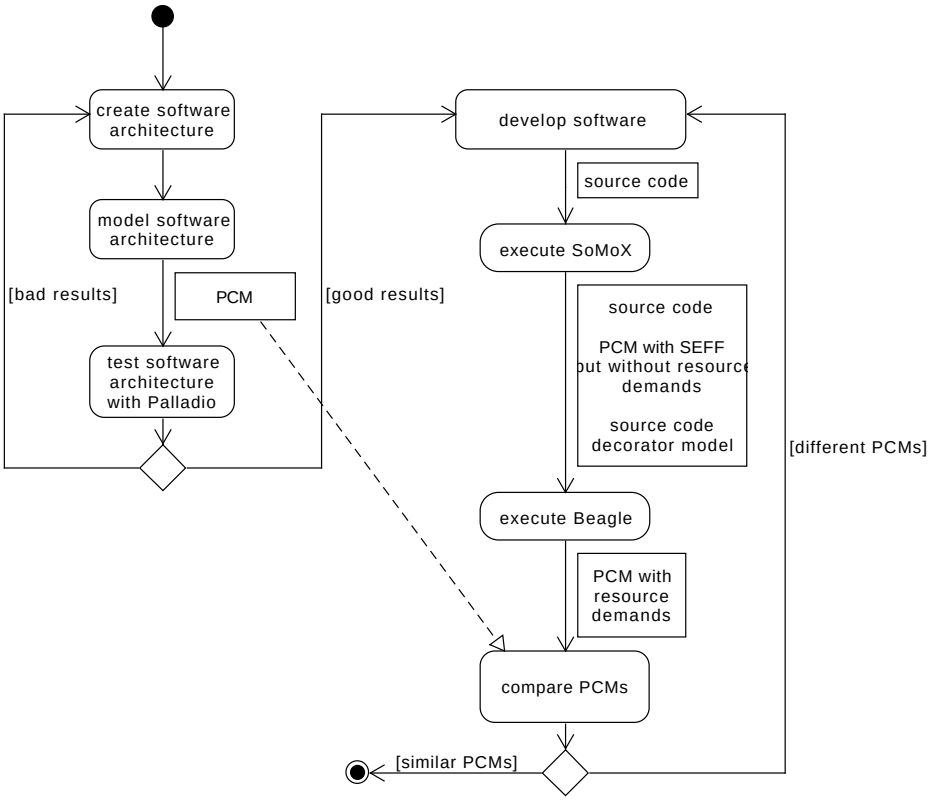

Figure 2.4: A typical workflow when using Beagle to verify the implementation (/A40/).

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 PCM source code decorator.

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

/E40/ Users should not run other programs while Beagle is running as this disturbs the measurement. To receive optimal measurements, Beagle should run on a dedicated server (See also /C130/).

3.1 Component Model

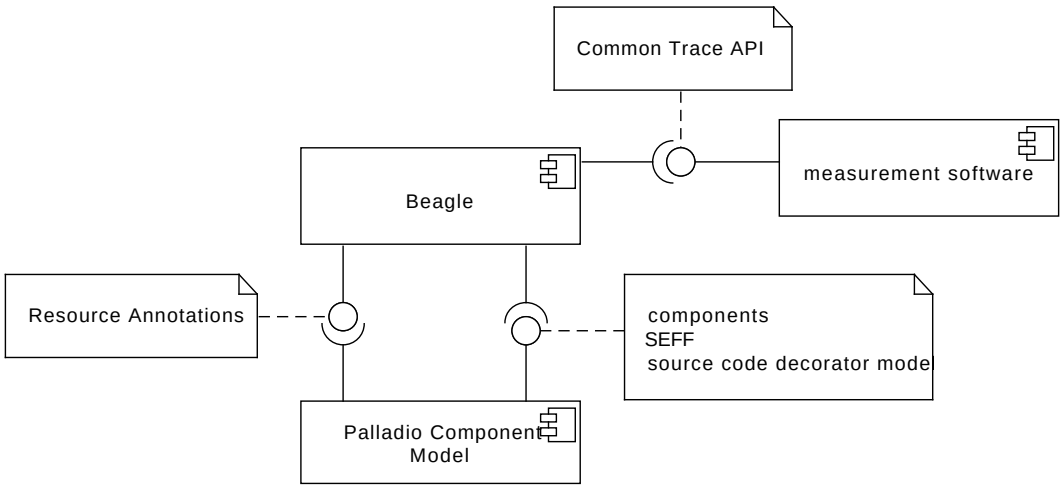

Figure 3.1: Beagle and its interaction with other software

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’ SEFFs to the parts in the source code implementing them. They must be modelled in the input PCMs source code decorator.

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' CPU resource demands.

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. 13), 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. Correctly determining these sections assures the measurement results do not depend on the measured component's assembly context.

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

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

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

/F50/ Beagle provides a functionality to stop measurements by an adaptive timeout when enabled. This means that it aborts a measurement when it exceeds a certain period of time. This timeout is adapted based on previous runs with the same or similar arguments. It is increased if these previous runs took a long time (as it is expected that these measurements will take a long time, too) and decreased if they took a short time to answer a request.

/F60/ Users can disable the adaptive timeout described in /F50/ and replace it with a set timeout or disable the timeout entirely.

Optional

/OF10/ Beagle approximately determines coherences between components' interface parameters and their resource demands. This enables Beagle to find resource demands independent from the measured component's usage context.

/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. This enables Beagle to express them independent from the measured component's usage context.

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

/OF50/ Beagle determines /OF40/ depending on the component's interface parameters. This enables Beagle to express them independent from the measured component's usage context.

/OF60/ Beagle runs benchmarks on hardware systems in order to provide information to make its results transferable: Using the benchmark information, measurement results can be transferred between different hardware systems when simulating in Palladio. This insures the independence from the component's deployment context.

5.2 Control

Optional

/OF100/ Users may choose whether Beagle will analyse the entire source code or only parts of it.

/OF110/ Users may choose to re-analyse the source code or parts of it in order to either gain more precision or to reflect on source code changes.

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

/OF130/ Users may pause and resume a measurement. Pausing causes all analysis activity to stop. Resuming continues the analysis from the beginning of the measurement it was taking when it had been paused.

/OF140/ Users may resume a paused measurement (/OF130/) even if Beagle had been closed after pausing it.

/OF150/ Beagle asserts that no input artefact (p. 14) has been changed between pausing (/OF130/) and resuming (/OF140/) an analysis 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. Therefore Beagle assures approximately constant measurement conditions, e.g. by heating up the CPU with load before starting measurements when resuming.

/OF170/ 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/ Beagle stores all its results in the software's PCM ("result PCM", see p. 13).

/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 measured components' internal actions' resource demands.

/F240/ Beagle does not remove any information from the input 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

In order to be as independent as possible and to provide good QoS and user experience, Beagle must fulfil the following requirements:

6.1 Dependencies

Mandatory

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

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

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

Optional

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

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

/OQ30/ Beagle shall handle any error caused by the measured software (uncaught exceptions, uncaught errors, calls to `System#exit`, or other unexpected termination of the software's process(es)). This means that neither will Beagle crash because of such errors nor will other measurements be affected by them.

/OQ40/ Beagle does not modify the provided source code files.

6.2 User Interface and Experience

Mandatory

/Q100/ Beagle is implemented as an Eclipse plug-in. Since both Palladio and its extensions are Eclipse plug-ins, this ensures good usability.

/Q110/ Beagle uses native Eclipse features for its GUI.

/Q120/ 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 so users do not need to provide additional information after SoMoX has been started. If Beagle requires more information than SoMoX provides, users can already submit it while configuring SoMoX.

/OQ120/ Beagle reports its progress to the user.

6.2.1 GUI Model

Users have several options to launch the analysis:

1. To analyse the entire 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 out and a description stating why the analysis is not possible is shown when a user tries to start it.
4. If /OQ100/ is implemented, users have the option to automatically start the analysis with Beagle after SoMoX has finished when launching the latter.

When users launch the analysis, they are presented a window where they can adapt certain Beagle settings:

1. If /OF60/ is implemented, users may adapt the default timeout.
2. If /OF120/ is implemented, the connection to the measurement machine can be set up.

3. If /OF60/ is implemented, users may choose to additionally benchmark their hardware system.

If /OF60/ is implemented, Beagle also provides a button for benchmarking the hardware without running any analysis.

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

If /OF130/ is implemented, there is a button for pausing the analysis. If it is paused, this button changes to a resume button. If /OF140/ is implemented and the users choose to close Eclipse, a dialogue allowing them to resume the analysis appears every time they launch Eclipse. This dialogue also offers the options to disable the dialogue for the future and to abort the analysis and drop the data collected to this point. Additionally, each context menu with the entry “Analyse with Beagle” also has another entry called “Resume Latest Beagle Analysis” allowing the user to resume the analysis.

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

7 Test Cases

7.1 Functionality

Beagle has to correctly interact with the interfaces defined in /E10/, /E20/ and /E30/. However, most tests must not include third party software but provide artificial input data instead. Doing differently could result in testing third party software, or, even worse, in not detecting errors and failures because they are compensated by other software. Therefore, tests in this section always premise test-provided input artefacts. Integration tests are described in the next section. Please note that /Q30/ and /OQ20/ are implicitly tested by the tests described below.

Mandatory

/T10/ Assert that Beagle 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 gracefully.

/T20/ Assert that Beagle discovers all sections needed for measurement and that they are correct. A part of this can be implemented by checking if all code sections (of the measured part) have been executed. Tests /F10/.

/T30/ Assert that Beagle correctly commissions measurements of the detected code sections through the CTA. Tests /F20/, /F30/ and /F40/.

/T40/ Assert that all all measured resource demands are added to the result PCM (which is read from where Beagle wrote it to) by comparing it with a manually created PCM instance. This includes to assert that the result PCM is valid. Tests /P10/, /P20/, /F200/, /F210/ and /F230/.

/T50/ Assert that all information found in the input PCM can be found in the result PCM. Tests /F240/.

/T60/ Assert that Beagle stops measurements after timeout. Provide input source code which does not terminate, define a timeout, and assert that Beagle terminates after that timeout, but not sooner. Additionally assert that this timeout can be turned off. Tests /F50/ and /F60/.

Optional

/OT10/ Assert that Beagle detects invalid input (e.g. if the PCM source code decorator does not fit to the code) and does not crash but responds to it in an acceptable way.

/OT20/ Assert that Beagle does not crash if the provided source code kills its process(es) in any possible way. Tests /OQ30/.

/OT30/ Assert that approximate coherences between input parameters and resource demands are annotated in the result PCM. This test can be extended to match Beagle's capabilities by providing source code with arbitrarily complex resource demand functions. Tests /OF10/.

/OT40/ Assert that the result PCM contains approximations of the probability of branches in SEFF conditions to be taken and numbers of repetitions in SEFF loops. Optionally assert that these approximations are expressed dependently on input parameters. Like /OT30/, this test can be extended to match Beagle's capabilities by providing source code with arbitrarily complex relations between input parameters and repetitions of SEFF loops, or branches taken in SEFF conditions. Tests /OF20/, /OF30/, /OF40/, /OF50/ and /OF200/.

/OT50/ Assert that it is possible for users to decide whether the whole source code or only parts of it are analysed. Start several runs for different parts in given source code and determine the different parts which have to be tested. Assert that all other tests concerning the result PCM pass for the results of each run. Tests /OF100/.

/OT60/ Assert that users are able to re-measure source code by measuring the same source code several times. Assert that the individual results are united correctly in the result PCM. Tests /OF110/.

/OT70/ Assert that pausing and resuming measurements works as expected by testing the same source code without pausing and several different numbers of pauses. The result PCM has to be the same (because artificial input data is provided, which is not exposed to environmental influences). Optionally, close Beagle or modify input artefacts between pausing and resuming. Tests /OF130/, /OF140/, /OF150/ and /OF160/.

/OT80/ Assert that Beagle shuts down the computer if requested in a manual test. Run a measurement and activate shutting down and then check if it worked and the results were saved. Tests /OF170/.

/OT90/ Assert that running measurements over a network works exactly like running measurements locally by running a reasonable number of the above tests on a setup that measures over a network. Tests /OF120/.

/OT100/ Assert that the input source code was not changed after Beagle has run. Tests /OQ40/.

7.2 Integration

Mandatory

/T200/ Assert that Beagle works on a system with only the software specified in /Q10/. A new system has to be set up with only these software applications and a reasonable number of the functionality tests have to be run on it (may for example be achieved by running all tests on a continuous integration service). Tests /Q10/.

/T210/ Assert that Beagle works with different software through the CTA. Test Beagle with Kieker and other measurement software. Tests /F40/ and /Q20/.

/T220/ Assert that the result PCM can be read by a Palladio installation without Beagle by opening it on such a system. Tests /F220/.

Optional

/OT200/ Assert that Beagle works for different operating systems and hardware by running a reasonable number of integration tests on different systems. Tests /OQ10/.

/OT210/ Assert that it is possible to run Beagle automatically after SoMoX has finished by running a reasonable number of functionality tests with Beagle being configured with and launched after SoMoX. 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 (considering 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 wanted to be analysed for their performance.

/C20/ The measured software system has a constant, deterministic runtime for a fixed configuration of input parameters when ignoring influences of the hardware, the operating system, and the error of measurement. This will be the case for most software systems. The fact that users try to measure the software system when using Beagle implies they expect it to behave in such a way.

/C30/ The input artefacts (see p. 13) are of integrity. 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 reverse-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.

/C40/ When using JaMopp, the source code's Java version is probably restricted by it. Making Beagle independent from JaMopp can be useful for future projects since they then would be able to use functions of newer Java versions in the software system to be tested.

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, as well as further measures.

/C110/ Beagle must ensure the transferability and scalability of its measurement results across different hardware platforms, in order to abstract them from their deployment context. 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. /OF60/ already addresses this.

/C120/ As Beagle should be able to measure specific components, other components of the software which are called during the measured component's execution may be desired to be mocked (especially those which take long time to return or do not return at all, e.g. a GUI). Mocking a component might be very effortful or even impossible. If this cannot be solved, Beagle may require users to provide a test bed in which the measured components can be executed.

/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 simultaneously. This could, however, influence Beagle's measurement results. If a considerable impact on measurement results is recognised, strategies to avoid them may be developed. These may including prompting the user to close certain applications. Users will likely be advised to provide a dedicated machine to run the measurements on.

/C140/ Beagle aims to parameterise 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 will probably 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 parameterisation at all. The genetic programming approach (/C150/) might help solving this, too.

/C150/ This projects focuses on measuring using dynamic analysis tools that provide their results through the CTA. [Krogmann, 2011], however, describes an approach combining different sources of resource demand data using genetic programming. It aims to combine their advantages and would enable Beagle to create a more accurate model of resource demands. Other performance measurement techniques (such as ByCounter) could be integrated. Whether this approach can be implemented in the scope of this project, needs further investigation and consideration.

9 Models

9.1 Scenario 1

EmmaSun[1], a Java-based online shop is running on a middle-class web server. During the first few years the software system was able to deal with almost 99.8% of incoming requests and orders quite well and without any noticeable delay. After an enormous expansion since the last year, the number of users is currently growing by about 5% per 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 individual applications. Unfortunately, the software system is based on an early design that has grown over the years with missing documentation in many cases. The effort to completly re-write it is unbearable. The code also scales bad, so buying new servers will not solve the problem either. The only solution seems to re-analyse the software's source code and architecture to hopefully find the bottlenecks that can be repaired with least effort. EmmaSun's developers have heard of Palladio and think it could serve them well to overcome their issues. Unfortunately, modelling all existing code is such a huge task that the management is reluctant to make this step.

At this point, Beagle and SoMoX come into play. The team of software architects that was commissioned by EmmaSun start to reverse-engineer a complete PCM instance modelling all software components and their SEFFs using SoMoX. In conjunction with Kieker, Beagle is then used to conduct measurements on the software's components, adding resource demand information to the PCM. After less than two days, the team is able to analyse its software with Palladio and run performance predictions for various approaches of improvement. The analysis reveals an architectural violation of certain software components, which leads to a a huge amount of inter-component calls throughout various hierarchical layers.

After revising several improvement approaches, EmmaSun's software architects decide to add an extra cache which can store the results of most external calls and makes them available almost immediately. predictions suggest that small changes in the software's architecture adopting these improvements will lead to a much better performance. The software architects agree to implement the new design.

[1] All characters and organisations appearing in this work are fictitious. Any resemblance to real persons or companies, living or dead, is purely coincidental.

After a two weeks, EmmaSun can already publish first changes that improve the shop's performance. The development team continues to use Palladio to model and plan their software's architecture, leading to further improvements in its code quality and QoS.

9.2 Scenario 2

Two years later, EmmaSun emerged to be an established and much-used online shop. The last years were busy and EmmaSun constantly hired new developers and deployed its software on an evergrowing cluster of servers. Because of their good experience with Palladio, EmmaSun's software architects never stopped to model their software using it. Thus, the software's architecture became more and more sophisticated, increasing EmmaSun's QoS along the way.

Today, EmmaSun's managers decided to offer a new, ground-breaking feature: Automatic audio conversion. Any audio piece, no matter whether it is sold on CD, tape, or LP, will automatically be offered to users in various digital music formats, from high quality FLACs to small 128 kbit/s MP3s. While the management is not, EmmaSun's development team is fully aware of the implications on performance this feature will have. They decide that careful planing is crucial to offer a reliable and fast conversion service.

As usual, the software architects start to model the new feature's sub-architecture and components in the PCM. They cooperate continuously with the system deployers to coordinate the necessary hardware changes. After that, the component developers start to implement the new components. They soon realise that there are multiple ways to implement the conversion engine. Especially, there are different ways to parallelise the task. Different developers come up with different approaches, all having their advatages and disadvantages. The development management soon realises that the effective performance will depend highly on the components' usage and deployment and cannot be predicted by simple measurements.

To determine the best solution, the developers start to implement a prototype for each approach. The software architects then import each of these prototypes into Palladio using SoMoX and Beagle, like they did two years ago. In Palladio, they are able to simulate the load they think the system will face on the new servers planned for purchase. Using the predictions, they are able to determine which approach will perform best for the planned context. The development team starts to implement it.

Four months later, the new feature is almost ready to be launched. Most code has been written and basic functionality has been asserted. The component developers are mainly fixing minor bugs while the system deployers prepare to purchase and install the newly required hardware. Meanwhile, the software architects import the new code into Palladio using SoMoX and Beagle to assure the software system will

hold the predicted QoS. When analysing the imported model, they recognise significant differences between the initial predictions for the prototype and the ones made for the actual implementation. The necessity to check copyright violations and content integrity as well as making backups and reporting progress to the user resulted in more network usage by the conversion component than predicted. Fortunately, this bottleneck can be fixed by using better network infrastructure hardware.

When the new conversion feature is implemented, users are astonished by its speed. EmmaSun's largest competitor, Million Shopping[2], releases a similar feature soon after. But as users start using it, it becomes slower and less reliable over time, resulting in unsatisfied customers and sometimes even failed transactions. EmmaSun's services, on the other hand, prove to be reliable and scale very well. A year later, EmmaSun will call the feature one of the main reasons they became the world's leading online shop.

[2] All characters and organisations appearing in this work are fictitious. Any resemblance to real persons or companies, living or dead, is purely coincidental.

Terms and Definitions

assembly context

the components in conjuntion with a component. Specifies which components provide the component's required interfaces.

Beagle

"**BE**haviour **A**nalysis using **G**enetic **L**earning and **E**volution". Approach for dynamic analysis of source code in order to find its behavioural attributes developed in [Krogmann, 2011]. This project aims to implement Beagle.

ByCounter

tool that instruments Java bytecode and executes it in order to count how often each method and Java Byte Code instruction is called. The resulting counts may serve as fine-grained, deployment-independent information about the measured code's resource demands [Kuperberg et al., 2008].

Common Trace API

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

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.

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." [Koziolek et al., 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]

component-based software

a software constituted of components.

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.

deployment context

in which environment a component runs. Specifies which resources are available to the component. Includes information like the performance of the hardware the component runs on but also information about the software resources like the virtual machine or thread pools.

internal action

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

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.

Kieker

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

A measurement software Beagle aims to support.

layer

describes conceptual separation in software.

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.

Palladio

an approach to the predict QoS properties of component-based software architectures with a special focus on performance properties.

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]

PCM source code decorator

realises links from the source code to the elements in the PCM and the other way round. [Krogmann, 2011]

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].

quality of service

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

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.

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.

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.

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.

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]

software architecture

the high-level structure and design of a software system as well as the discipline of creating and documenting these.

SoMoX

"**So**ftware **Mo**del e**X**tractor", a Palladio plugin for static code analysis to re-engineer a software's architecture from its source code developed in [Krogmann, 2011]. Constructs a PCM instance including the reconstructed components and their SEFF.

system

the useful whole created from diverse parts. A system (usually) reflects the organizational structure that built it. (Conway's law) [Conway, 1968]

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]

usage context

how a component is used. Includes the number, frequency and distribution of calls made the the component's services.

Bibliography

[Conway, 1968] Conway, M. E. (1968). How do committees invent? *Datamation*.

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

[Koziolek et al., 2007] Koziolek, H., Happe, J., Becker, S., and Reussner, 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.

[Kuperberg et al., 2008] Kuperberg, M., Krogmann, M., and Reussner, R. (2008). ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations. In *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation , Budapest, Hungary, 5th April 2008 (ETAPS 2008, 11th European Joint Conferences on Theory and Practice of Software)*.

[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.