

# QoS Data Collection: An Approach to Assist Predictable QoS Behavior Modeling in Component Based Development

Jia Zhou, Kendra Cooper, I-Ling Yen  
Department of Computer Science  
University of Texas at Dallas  
{jxz023100, kcooper, ilyen}@utdallas.edu

## Abstract

*This paper presents a Quality of Service (QoS) data collection approach to assist predictable QoS behaviors modeling in component based development. The approach has three steps. The first step is test driver generation. The second step is test data generation. The third step is QoS data measurement. QoS attribute data are collected for a library of 338 numerical analysis components, executed on three different platforms (windows, unix, linux).*

## 1. Introduction

Embedded software continues to become more and more complex due to the growing sophistication of modern applications. As can be expected, embedded software developers are going to face a great challenge in the development of embedded software over the years. They urgently require effective development approaches.

Component based development (CBD) is a promising approach to promote software development quality. It can benefit traditional software development as well as embedded software development. However, applying CBD to embedded software development faces additional challenges due to the stringent Quality of Service (QoS) requirements of embedded systems. It is crucial for embedded systems to consider QoS attributes, such as timeliness, memory limitations, output precisions, battery constraints. Thus, CBD techniques for embedded software development must integrate QoS analysis into the entire development process. An example of such CBD techniques is ORES [1][2], which integrates a QoS prediction model to assist the selection, adaptation, and integration of components.

Frequently, multiple components in a component repository may implement the same functionality with different QoS tradeoffs. Moreover, some of these components may be configurable, i.e., some of the program parameters of a component can be configured to achieve different QoS tradeoffs [3][4]. It can be computationally intensive to use an exhaustive search approach to determine the most suitable set of components

and the best parameter settings. Thus, an efficient and effective decision making model is needed for QoS analysis in a CBD technique for embedded software development.

A number of different techniques have been proposed to build QoS models from composition specifications and perform QoS analysis based on the models [5][6]. For example, a QoS prediction model [6] based on synchronous data flow (SDF) has been presented. This model formulates the QoS analysis as a multi-objective optimization problem and uses an evolutionary algorithm to obtain the pareto-optimal solutions to determine suitable component selections and parameter settings.

A common assumption underlying these QoS models is that QoS behaviors of individual components are predictable. Consequently, data sets for QoS attributes of each component must be available to predict a system's QoS behaviors based on these QoS models.

Frequently, data sets are acquired through non-functional testing of the components. However, software testing community still focuses on testing the functional aspect of components, which analyze the characteristics of component-based software and suggest issues in testing and maintaining component-based systems [7][8][9]. Although a number of techniques have been proposed for non-functional testing in recent years, they are limited to the areas of performance [10][11], usability [12][13], and security testing [14][15]. Testing of other QoS attributes (e.g., memory, quality) are not well addressed.

This research proposes a non-functional testing process to address this requirement. Data sets for time, space, and quality attributes can be collected through the process and used to predict the QoS behaviors of individual components or a collection of integrated components. Furthermore, these data sets are publicly available. Other researchers and developers can reuse these data sets to perform their own QoS analysis. Additionally, if the data sets for a specific set of components are not existent, then each researcher or developer can reuse the proposed non-functional testing process to collect their own data sets. Such reuse behavior avoids a lot of time, effort, and expertise to be spent in creating an alternative testing process.

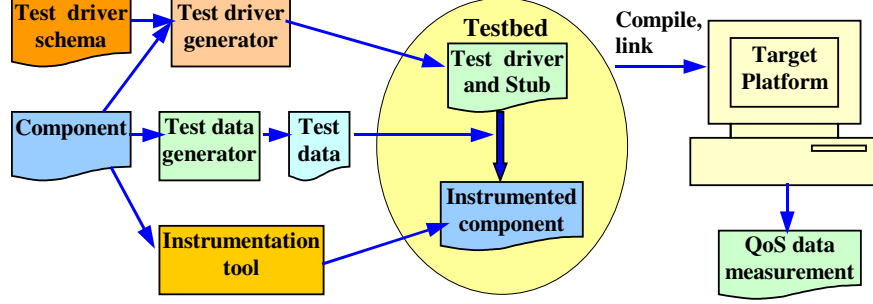


Figure 1. QoS data collection process

This paper is structure as follows. The QoS data collection technique is introduced in the next section. The entire process of QoS data collection is first discussed followed by the presentation of case study. Conclusion and future work are presented in Section 3.

## 2. QoS data collection

The QoS data collection process consists of three steps (refers to Figure 1). First, a testbed is established to simulate an environment for running a component and simulating missing subroutines. The testbed consists of a test driver and/or stubs in order to test an individual component. Stubs are used to simulate subroutines called by the component under test. A test driver contains the code to establish an environment and invoke the component under test. The test driver is generated automatically from a test driver schema. The test driver schema specifies the generic test execution step, which describes a pattern that performs the steps necessary to execute a test case. The schema is defined once per test environment. The schema algorithm performs initialization and then loops through each test to loads inputs, calls the component under test, and then retrieves and stores the actual test outputs.

Second, test data are generated for input parameters. If the component contains configurable parameters, then test data are also generated for configurable parameters of the component. Currently, a random approach is used to generate test data for input parameters and configurable parameters. A rule-based approach [3][4] is used to identify configurable parameters of a component. This approach uses compiler techniques to build a parse tree for the source code of a component, extracts facts from the parse tree, and applies identification rules on these facts to identify configurable parameters.

Finally, the component is instrumented to add in auxiliary code to collect time and space data. The test bed program (test driver, instrumented component and stub) is then executed using each test datum; data sets for three QoS attributes include time, space and quality are measured. Currently, only the average behaviors (i.e., not worst case) of these three QoS attributes are measured in

the process due to the limitation of the random test data generation method. The prototyped tool set for the QoS data collection can be found at [17].

### 2.1. Case study

An empirical study of the QoS data collection technique has been conducted on a library of 338 numerical computation components developed in C language (Currently, the QoS data collection technique only handles C programming language). These components handle various aspects of numerical computation such as computing solutions of linear algebraic equations, calculating integration of functions, generating random numbers, finding root and nonlinear sets of equations, etc.

The biconjugate gradient method is a numerical algorithm for solving the  $N \times N$  linear system  $A \cdot x = b$ . Given a system, the algorithm starts with an initial guess  $x_1$  for the solution. Choose  $\bar{r}_1$  to be the residual  $\bar{r}_1 = b - A \cdot x_1$ , choose  $\bar{p}_1 = \bar{r}_1$ , and set  $p_1 = \bar{r}_1, \bar{p}_1 = \bar{r}_1$ . The algorithm repeatedly refines the estimates  $x_{k+1} = x_k + \alpha_k p_k$  by carrying out the recurrence (1) until reaching the stopping criterion  $F < tol$  where  $F$  is a formula and  $tol$  is an error tolerance. Here,  $tol$  is a configurable parameter whose value affects the execution time and output quality.

$$\begin{aligned} \alpha_k &= \frac{\bar{r}_k \cdot \bar{r}_k}{\bar{p}_k \cdot A \cdot p_k}, & r_{k+1} &= r_k - \alpha_k A \cdot p_k \\ \bar{p}_{k+1} &= \bar{p}_k - \alpha_k A^T \cdot \bar{p}_k, & \beta_k &= \frac{\bar{r}_{k+1} \cdot \bar{r}_{k+1}}{\bar{r}_k \cdot \bar{r}_k} \\ p_{k+1} &= r_{k+1} + \beta_k p_k, & \bar{p}_{k+1} &= \bar{r}_{k+1} + \beta_k \bar{p}_k \end{aligned} \quad (1)$$

Given the source code of the algorithm (shown in Figure 2), the time and quality attributes can be reconfigured by adjusting the initial values of the macro identifiers  $ITOL$  and  $TOL$  (i.e.,  $tol$  in ' $F < tol$ '). The values of  $ITOL$  and  $TOL$  are passed to input parameters

*itol* and *tol* of routine *linbcg*. The value of *ITOL* determines the formula *F* used in the routine. If *ITOL*=1, iteration stops when the quantity  $|A \cdot x - b|/|b|$  is less than the tolerance *tol*. If *ITOL*=2, the required criterion is  $|\tilde{A}^{-1} \cdot (A \cdot x - b)|/|\tilde{A}^{-1} \cdot b| < tol$ . If *ITOL*=3, the routine uses its own estimate of the error in *x*, and requires its magnitude, divided by the magnitude of *x*, to be less than *tol*. The setting *ITOL*=4 is the same as *ITOL*=3, except that the largest (in absolute value) component of the error and largest component of *x* are used instead of the vector magnitude [16].

```
#include ...
#define ITOL 1
#define TOL 1e-9
#define EPS 1.0e-14

void linbcg(unsigned long n, double b[], double x[], int itol, double tol,
            int itmax, int *iter, double *err)
{
    unsigned long j;
    double ak,akden,bk,bkden,bknum,bnorm, dxnorm,xnorm,zm1norm,znorm;
    double *p,*pp,*r,*rr,*z,*zz;
    // Skip the code for allocating memory for p,pp,r,rr,z,zz
    // Skip the code for calculating initial residual.
    while (*iter <= itmax) { // Main loop.
    // Skip the code for calculating coefficient bk and direction
    // vectors p and pp.
    // Skip the code for calculating coefficient ak, new iterate x, and
    // new residuals r and rr.
        if (itol == 1) // Check stopping criterion.
            *err=snorm(n,r,itol)/bnorm;
        else if (itol == 2)
            *err=snorm(n,z,itol)/bnorm;
        else if (itol == 3 || itol == 4) {
            // Skip the code for calculating err
        }
        if (*err <= tol) break;
    }
    // Skip the code for releasing memory
}
```

**Figure 2. Biconjugate gradient method**

## 2.2. Experimental design

**Independent variables.** The experiment is conducted on three different platforms including Unix, Linux, and Windows. The Unix platform runs SunOS 5.9. It equips a 500MHz UltraSPARC Iii processor, 128MB of memory, and 7200RPM of hard drive. The Linux platform runs Linux Fedora Core 4. It equips a 2.66GHz Intel Pentium 4 CPU, 1GB of memory, and 7200RPM of hard drive. The Windows platform runs Windows XP. It equips a 2.8GHz Intel Pentium 4 CPU, 1GB of memory, and 7200RPM of hard drive.

There are 338 components from a numerical analysis library used. Each component is executed a total of 10,000,000 times: 100 values for the configurable

parameters and 100 values for the input parameters are randomly selected; each component is executed 1000 times for each test datum of the configurable parameters and input parameters. The configurable parameters are identified using the approach presented in [3][4].

For example, in the biconjugate gradient method, two configurable parameters are identified for time-quality tradeoff:

1. the configurable parameter *ITOL*. It is of integer type. Its value is selected in the domain of [1,2,3,4]; and
2. the configurable parameter *TOL*. It is of double type. Its value is selected in the domain of [0.1 to 1e-14].

**Dependent variables.** The dependent variables are the QoS attributes:

1. the time attribute. Average response time is measured. The unit of measurement is milliseconds. For the biconjugate gradient method component, the calculation is:

$$\overline{\text{time}_{\text{cps},I}} = \frac{\sum_{i=1}^{100} \text{time}_{\text{cps},I}}{100}$$

where  $\overline{\text{time}_{\text{cps},I}}$  stands for the average response time,  $\text{time}_{\text{cps},I} = \text{entry time}_{\text{cps},I} - \text{exit time}_{\text{cps},I}$ , *cps* stands for configurable parameters, *I* stands for input parameters, and 100 stands for 100 test data for input parameters.

2. the space attribute. Average memory usage is measured. The unit of measurement is bytes. Measurement of space attribute only considers statically allocated memory (i.e., array) and dynamically allocated memory (i.e., memory allocated through routine “malloc”, “alloc”, etc.). The stack, heap, and other memory allocated by system for execution are not counted since their sizes depend on compiler and platform. For the biconjugate gradient method component, the calculation is:

$$\overline{\text{memory}_{\text{cps},I}} = \frac{\sum_{i=1}^{100} \text{memory}_{\text{cps},I}}{100}$$

where  $\overline{\text{memory}_{\text{cps},I}}$  stands for the average memory,  $\text{memory}_{\text{cps},I} = \text{static memory}_{\text{cps},I} + \text{dynamic memory}_{\text{cps},I}$

3. the quality attribute. Average output quality is calculated. The unit of measurement varies for different applications. For example, it may be the

compression ratio for a compression program or the error for a numerical computation program. For the biconjugate gradient method component, the calculation is:

$$\overline{\text{quality}}_{\text{cps},I} = \frac{\sum_{l=1}^{100} \text{quality}_{\text{cps},I}}{100}$$

where  $\overline{\text{quality}}_{\text{cps},I}$  stands for the average quality,

$$\text{quality}_{\text{cps},I} = \left| \text{error}_{\text{cps},I} \right|.$$

**Analysis strategy.** The analysis strategy has three steps. First the measured data are summarized and plotted in figures. Then the fitting surfaces are calculated by using interpolation techniques and plotted in the figures. Finally, a comparison among three platforms is made.

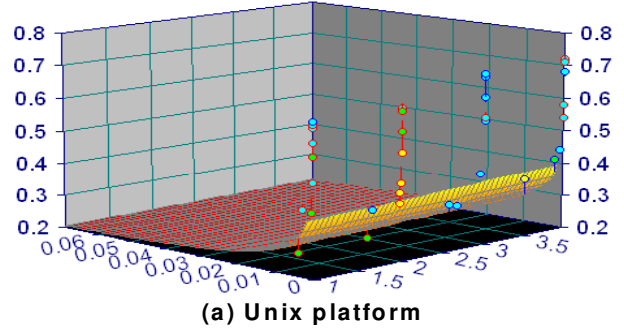
### 2.3. Results and analysis

The QoS data sets measured for the example component on three platforms has been collected. Due to the limitation of space, this section only presents the response time results measured for an example component in the library – biconjugate gradient method. The data sets for other components in the library can be found at [17].

The time data varies on different hardware and OS. Figure 3 show the measured time data on three platforms.

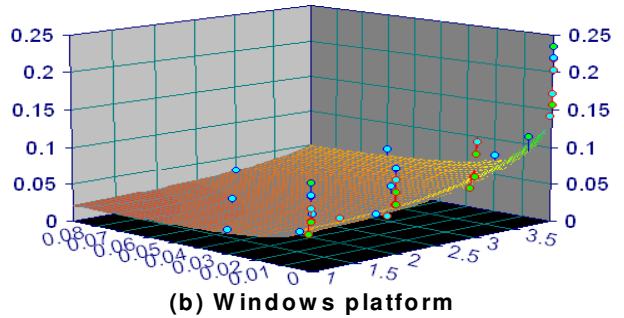
The x-axis in each figure refers to values of configurable parameter *ITOL*. The y-axis in each figure refers to values of configurable parameter *TOL*. The z-axis in each figure refers to data of execution time in milliseconds. Fitting surfaces are computed from the measured raw data for each platform. These surfaces give the approximate function representation for the relationship between the time attribute and the configurable parameters *ITOL* and *TOL*.

As expected, the fitting surfaces for the three platforms demonstrate the similar trend in different scales. The Linux platform is the fastest among the three platforms; the Windows platform is moderate; the Unix platform is the slowest. The response time on the Unix platform is approximately 7 times the response time on the Linux platform; the response time on the Windows platform is approximately 2 times the response time on the Linux platform. The trend shows how the configurable parameters *ITOL* and *TOL* would impact the response time attribute. As can be seen from the figure, decreasing *TOL* yields increased processing time. This trend is especially prominent when *TOL* is less than 0.01. Increasing *ITOL* also yields increased processing time. However, the relationship between response time and *ITOL* is not as strong as the relationship between response time and *TOL*.



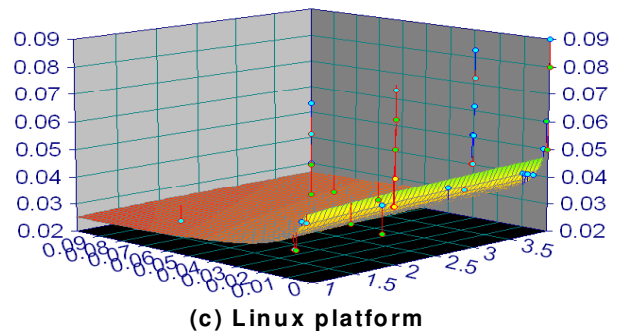
Fitting equation:

$$z^{-1} = 1.5538049 - 0.098182414 (\ln x)^2 - 4.7309546 y^{0.5} \ln y$$



Fitting equation:

$$\ln z = 2.3893958 + 0.013604508 x^3 + 2.0365962 y^{0.5} \ln y$$



Fitting equation:

$$z^{-1} = 13.227488 - 1.5137949 \ln x - 36.146305 y^{0.5} \ln y$$

**Figure 3. Time vs. configurable parameters (ITOL and TOL)**

The analysis results of the QoS data can help developers determine how to adjust the configurable parameters in components to provide predictable QoS behaviors as execution conditions change, either at the development time or during the run time. Changes in development conditions include: (1) the Stakeholders' changing requirements; (2) manufacture of similar products in a product line; (3) the need to deploy the

software on heterogeneous platforms. Changes in run time conditions may include: (1) high system load; (2) component failure; (3) hardware upgrade (e.g., memory upgrade). For example, consider an application using this component. The Stakeholders change the application's QoS requirements. They decide to port the application to the Linux platform. They also require that the average output precision (error) of the component is less than 0.00001 and the average response time of the component is less than 0.06 milliseconds. Given the fitting equation  $z^{-1}=13.227488-1.5137949\ln x-36.146305y^{0.5}\ln y$  of the measured time data and the fitting equation  $\ln z=1.0290003-0.096800597x^{1.5}+1.1875691\ln y$  of the measured quality data for the Linux platform, the values of the configurable parameters *ITOL* and *TOL* can be determined by using the evolutionary algorithm proposed in [6]. For this example, the algorithm finds that a good candidate is *ITOL* = 1 and *TOL* = 0.0002.

### 3. Conclusion and future research

This paper presents a non-functional testing approach for measuring components' QoS attributes including time, space and quality. The data can be used to predictably select and/or integrate components.

The QoS data collection approach has several advantages. The approach has a defined process, which can be automated with the assistance of tool support. Other researchers and developers can use the approach to measure their own QoS data. These QoS data sets, when publicly accessible, can avoid duplicate efforts and support the replication of experiments and the comparison of results.

The QoS data collection approach also has some limitations. Currently, the approach does not measure the worst-case QoS data due to the limitation of random test data generation method.

There are several research directions following this work. First, additional component examples are going to be studied for more empirical data. An AMR speech codec library is the next library to be investigated. Second, the approach is going to be extended to support components developed in other programming languages. Third, measurements of the worst-case QoS data, such as worst-case execution time, worst-case space, and worst-case quality are going to be investigated. Finally, measurements of other QoS attributes are going to be studied, such as power consumption and footprint.

### 6. References

[1] I-L. Yen, Latifur Khan, Balakrishnam Prabhakaran, Farokh B. Bastani, and John Linn, "An on-line repository for

embedded software", *Proceedings of IEEE ICTAL*, Dallas, Texas, November 2001, pp. 314–324.

[2] I-L. Yen, Jayabharath Goluguri, Farokh B. Bastani, and Latifur Khan, "A component-based approach for embedded software development", *Proceedings of IEEE ISORC*, Crystal City, VA, April 2002, pp. 402–410.

[3] J. Zhou, K. Cooper, I-L. Yen, and R. Paul, "Rule-Based Technique for Component Adaptation to Support QoS-based Reconfiguration", *Proceedings of ISORC*, Seattle, WA, May 2005, pp. 426–433.

[4] K. Cooper, J. Zhou, H. Ma, I-L. Yen, and F. Bastani, "Code parameterization for Satisfaction of QoS Requirements in Embedded Software", *Proceedings of ERSA*, Las Vegas, NV, June 2003, pp. 58–64.

[5] Q. Tran and L. Chung, "NFR-Assistant: Tool support for achieving quality", *IEEE Symp. On Application-Specific Systems and Software Engineering and Technology*, Dallas, Texas, March 1999, pp. 284–289.

[6] H. Ma, I-L. Yen, J. Zhou, and K. Cooper, "QoS analysis for component-based embedded software: Model and methodology", *Journal of Systems and Software*, September 2005, pp. 859–870.

[7] D.S. Rosenblum, "Adequate testing of component based software", Technical Report TR97-34, University of California at Irvine, 1997.

[8] M.J. Harrold, D. Liang, and S. Sinha, "An approach to analyzing and testing component-based systems", *Proceedings of ICSE International Workshop on Testing Distributed Component-Based Systems*, Los Angeles, CA, May 1999.

[9] J. Voas, "Maintaining component-based systems", *IEEE Software*, July 1998, 15(4), pp. 22–27.

[10] P. Puschner and C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs", *Journal of Real-Time Systems*, September 1989, pp. 159–176.

[11] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms", *Software Quality Journal*, 1997, 6(2), pp.127–135.

[12] N. Jacobsen, and K. Landauer, "A mathematical model of the finding of usability problems," *Proceedings of ACM INTERCHI*, Amsterdam, Netherlands, April 1993, pp. 206–213.

[13] N. Jacobsen, M. Hertzum, B. John, "The evaluator effect in usability tests", *CHI 98 conference summary on Human factors in computing systems*, Los Angeles, CA, April 1998, pp. 255–256.

[14] C. Pfleeger, S. Pfleeger and M. Theofanos, "A Methodology for penetration testing," *Computers and Security*, 1989, 8(7), pp. 613–620.

[15] G. Fink and M. Bishop, "Property Based Testing: A New Approach to Testing for Assurance", *ACM SIGSOFT Software Engineering Notes*, July 1997, 22(4), pp. 74–80.

[16] H.P. William, P.F. Brian, A.T. Saul, and T.V. William, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

[17] <http://www.utdallas.edu/~jxz023100/PROMISE06>