An approach to quantifying the run-time behaviour of Java GUI applications

Aine Mitchell, James F. Power [*][†]

Abstract

This paper outlines a new technique for collecting dynamic trace information from Java GUI programs. The problems of collecting run-time information from such interactive applications in comparison with traditional batch style execution benchmark programs is outlined. The possible utility of such run-time information is discussed and from this a number of simple run-time metrics are suggested. The metrics results for a small `CelsiusConverter` Java GUI program are illustrated to demonstrate the viability of such an analysis.

Index Terms: Collecting dynamic trace data from Java GUI applications. Quantification of run-time elements of GUI applications. Possible applications of run-time metrics.

1 Introduction

Today Graphical User Interfaces (GUIs) are a popular method of interacting with software. The rapid increase in their use is due to the fact they make the software easier to use from the perspective of the user. The extensive use of GUIs is consequently leading to them becoming increasingly complex. However, a drawback of using a GUI is it increases the overall complexity of the software.

The special characteristics of a GUI program suggest that traditional methods of statically evaluating its complexity may not be suitable. GUI programs, unlike conventional software, are event-based systems. The input to a GUI is some action as specified by the user, which then will modify the state of the GUI. Each action may depend on the effect of the previous action. A static analysis of the source code only measures what may happen when the program is executed whereas a dynamic analysis attempts to quantify what actually happens. Therefore how the user chooses to interact with the GUI will have no effect on the value obtained from a static metric. As the ability of static metrics to accurately predict the run-time complexity of any program is as yet definitively unproven, it may prove useful to evaluate a GUI in its operational environment in a variety of contexts.

A dynamic analysis of a GUI application may be problematic as GUIs cannot be executed in the batch style of a standard benchmark suite like SPEC. The execution of a GUI is interactive where the user's commands and the computer's responses are interleaved during a single run. It is therefore necessary to define a new methodology of collecting dynamic trace data for a Java GUI program.

2 Graphical User Interface Programs

A GUI is becoming an increasingly important part of many software applications. It makes software easier to use and recent studies have shown that developers are dedicating a larger portion of code

[*]Department of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland.

[†]Please address correspondence to ainem@cs.may.ie

Accepted for the Winter International Symposium on Information and Communication Technologies Cancun, Mexico, January 5-8, 2004

to GUI implementation. It is thought that as much as 60% of the overall source of a program can constitute the GUI [4].

A GUI is a hierarchical, graphical front-end to a software that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects and each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI [5].

A number of studies have discussed how GUI driven applications have characteristics different from traditional software [4,5]. GUIs are typically constructed using instances of precompiled library elements. A drawback of this approach is that it is not always possible to obtain the source code for these elements for further study. Also, a GUI takes input in the form of a sequence of events. There is usually a large number of permutations of these events that can lead to a large quantity of different GUI states. The GUI may need to be evaluated in a large number of these states for a comprehensive analysis. The GUI input event sequences are conceptually at a much higher level of abstraction than the code and therefore cannot be acquired from the code. Therefore a method of running the programs in a documentable and repeatable manner must be developed.

The most widely used technique to reproduce a GUI program's execution is through the use of record/replay tools [7]. A record/replay tool is a tool that captures user inputs as they are being performed on the application and stores them in a script format. These scripts can be played back at a later time to simulate the user inputs captured by them. There are a number of modes for the record/replay tools available. This study demanded a tool that could operate in widget mode. A widget is an element of a GUI that displays information or provides a specific way for a user to interact with the operating system and application. Widgets include elements like icons, pull-down menus, selection boxes or other such devices for displaying information and for inviting, accepting, and responding to user actions. A record/replay tool operating in widget mode will capture all the user activity on the widgets including mouse events and type in text events. The recorded script will consist of a sequence of events associated with the corresponding widgets. These scripts are replayed by identifying the widgets in the GUI and performing the event associated with it.

3 Work Done to Date

A preliminary study on formulating a number of run-time definitions for coupling and cohesion has been completed. These metrics were based on the static object-oriented metrics of Chidamber and Kemerer, which are designed to evaluate the external and internal complexity of a piece of software. The newly defined run-time metrics were applied to assess the quality of a number of Java programs from the Java Grande Forum and SPEC JVM98 benchmark suites. A summary of work to date is available from [6].

Results obtained showed that the run-time coupling and cohesion values for the benchmarks under consideration were different from those predicted from a static analysis of the code. It is suggested that valuable information such as the state-based behaviour, dynamic binding, frequencies of method invocation or instance variable access, which is not obtainable from classic static metrics, is provided from a run-time evaluation. The possible utility of this runtime information is also examined. It is suggested that it may prove useful for the detection of anomalies in the code and may also play a role in quantifying the effectiveness of software testing strategies. It is also suggested that they may prove useful in evaluating the external quality attributes of a program.

4 Dynamic Metrics

A number of simple metrics are defined here designed to give an approximation of attributes of a program such as its size, structure and memory use.

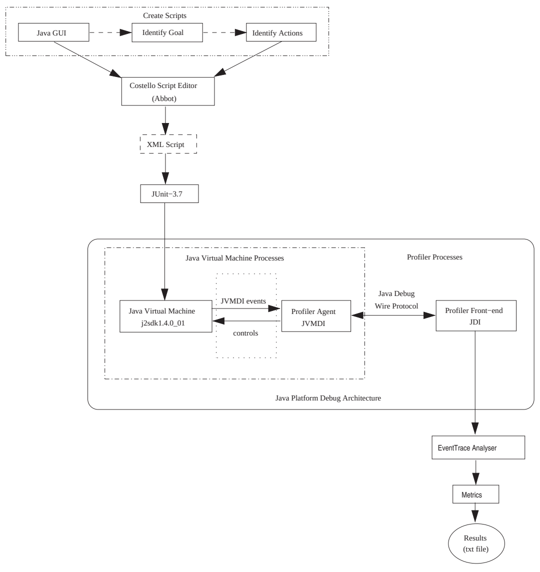

Figure 1: Our System for Collecting Run-time Metrics from Java GUI Programs

- **Size**
  meth.ob: The total number of public, protected and private methods called normalised by the total number of objects created. This can provide a crude measure of the size of a program.

- **Structure**
  exPubMet.Ob: The number of public methods called which are not from the calling class divided by the total number of objects created at run-time. This metric provides a crude measure of the level of coupling within a program at run-time. A class is deemed to be coupled to another class if it requests an attributes from that class. Therefore this metrics provides a measure of the external workings of a program. A high level of coupling within a program is not desirable as the principals of good object-oriented design desire that a class be as self sufficient as possible.

  priMet.ob: The total number of private methods called divided by the total number of objects created. This provides a crude measure of the level of cohesiveness within a program. Cohesion is a measure of how the tasks performed by a single module are functionally related. The rationale behind this measure is that the private methods are those that are involved in the internal working of the class and are therefore responsible for fulfilling public method contracts.

- **Memory Use**
  meth.inst: Measures the number of methods called per kilobyte (kbc) executed. It is computed as total number of methods, divided by (number of instructions executed / 1000). This will give an estimation of the memory use of the methods.

  ob.inst: The total number of objects created divided by the number of kbc executed. This provides a measure of the number of objects created per 1000 bytecode instructions executed. This metric gives an indication of how memory hungry the program is overall.

5 Experimental Platform

5.1 Design Objectives

The dynamic analysis of any program involves a huge amount of data processing. However, the level of performance of the collection mechanism was not considered to be a critical issue at this

XML Script Name	**Event Sequences**
A	Input $< 12 >$, Click $< Convert >$
B	Input $< -23 >$, Click $< Convert >$
C	Input $< 0 >$, Click $< Convert >$
D	Input $< 23 >$, Click $< Convert >$, Input $< 45 >$, Click $< Convert >$, Input $< 56 >$, Click $< Convert >$, Input $< -87 >$, $< Convert >$, Input $< 34 >$, $< Convert >$

Table 1: Event Sequences Specified in XML Scripts

time. It was only desirable that the analysis could be carried out in reasonable and practical time. It was however necessary to be able to collect a wide variety of dynamic information, therefore the collection mechanism had to be designed with a high degree of flexibility in mind.

5.2 Design

The metric data collection system consists of a number of parts. Firstly the Abbot-0.9.0 framework [1] is used in conjunction with the JUnit-3.7 [3] testing framework to create and execute scripts that simulate the execution of a Java GUI program. The Java Platform Debug Architecture (JPDA) [2] is responsible for generating the event traces, while a specially designed back-end `EventTrace` analyzer class carries out the processing of these traces. The final component of the collection system is a `Metrics` class, which is responsible for computing the values for the desired metrics.

The Abbot framework is a Java library that provides methods to reproduce user actions. It works by improving on the rudimentary functions provided by the java.awt.Robot class. Abbot contains a script editor called Costello which allows the recording of XML scripts in widget mode which simulate the actions of the GUI under consideration.

Junit, an instance of the xUnit architecture for unit testing frameworks, is used as the controlling harness for executing the scripts.

The test case scripts for the GUI's under consideration were manually constructed. Several of the most common goals of the GUI were intuitively identified. The sequence of events that the user might employ to achieve these goals were specified. These sequence of events became the test case inputs for the GUI. This method of identifying test cases for GUI programs is loosely based on the ideas proposed by [5].

Runtime trace information was obtained by utilising the JPDA. This is a multi-tiered debugging architecture contained within Sun Microsystem's Java 2 SDK version 1.4.0_01. It consists of two interfaces, the Java Virtual Machine Debug Interface (JVMDI), and the Java Debug Interface (JDI), and a protocol, the Java Debug Wire Protocol (JDWP). The first layer of the JPDA, the JVMDI, is a programming interface implemented by the virtual machine. The second layer, the JDWP, defines the format of information and requests transferred between the process being debugged and the debugger front-end which implements the JDI. The JDI, which comprises the third layer, defines information and requests at the user code level. It provides introspective access to a running virtual machine's state, the class, array, interface, and primitive types, and instances of those types. This was selected because of the ease with which it is possible to obtain specific information about the run-time behaviour of a program. It also has the advantage of being compatible with a number of commercial virtual machine implementations. However, it is currently not possible to obtain directly information about the state of the execution stack using this approach.

To overcome this problem the `EventTrace` analyser class, which is implemented in Java, carries out a stack based simulation of the entire execution in order to obtain information about the state of the execution stack. This class also implements a filter which allows the user to specify which events and which of their corresponding fields are to be captured for processing. This allows a high degree of flexibility in the collection of the dynamic trace data.

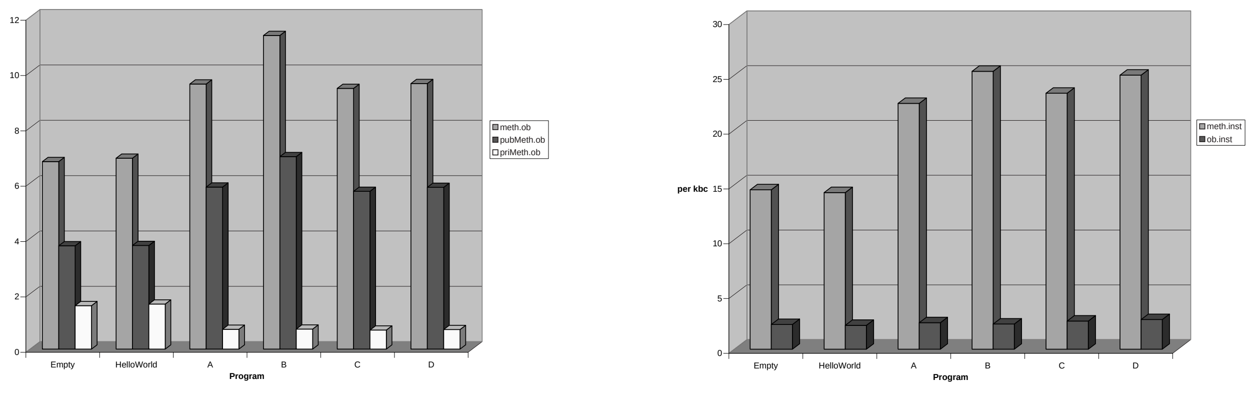

Figure 2: Dynamic Metric Results for Sample Programs

The required run-time measures are then calculated on the fly by a `Metrics` class which is also responsible for outputting the results in text format. The metrics to be calculated can be specified from the command line. The addition of the metrics class allows new metrics to be easily defined as the user need only interface with this class.

5.3 Analysis

Metric data was collected for a small Java `CelsiusConverter` program the function of which is to convert Celsius values to Fahrenheit with a number of test case inputs. An analysis was also carried out on the `empty` program and a simple `HelloWorld` program for the sake of comparison.

Firstly a number of the most obvious goals of the user of the `CelsiusConverter` GUI were identified. This is a relatively simple GUI which only allows the conversion of a positive, negative or zero Celsius values to Fahrenheit. The event sequences necessary to achieve these goals were specified.

A number of XML scripts were then created which exercised these goals as illustrated by Table 1.

6 Results

In this section we present some results from our analysis of a number of GUI applications. We stress that this study is still at a preliminary stage, and we present these results simply as an indication of some of the issues involved in this analysis.

The meth.ob metric gives a crude estimation of program size. Figure 2 deems the GUI program to be the larger as one would expect.

exPubMet.Ob gives an estimation of the level of coupling present in a program. From Figure 2 it can be seen that the GUI program has a greater external public methods called per object ratio than the benchmarks.

The priMet.ob metric shows that the `empty` and `HelloWorld` program devote a greater proportion of their method access to the internal working of their classes, than the GUI program. This is illustrated by the fact that they have a higher private methods per object ratio than the GUI.

The GUI program exhibited a higher method to kbc executed ratio than the `empty` or `HelloWorld` program for the meth.inst metric.

All the programs exhibited a similar value for the ob.inst metric, as they are all small, simple programs this is as expected.

7 Applications of this Study

An analysis of this kind may have a number of useful applications. Using this type of approach of it will be possible to measure how different test case inputs affect the internal properties of a program. This may be useful in the quantification of different software testing strategies and may also be useful in identifying code that may benefit from restructuring when refactoring.

Metrics of this type may be of use to the compiler community as they quantify different aspects of the dynamic behaviour of programs.

This approach will allow inheritance to be more closely examined to investigate how its use actually affects the runtime performance of a program.

It should also be possible to design metrics to quantify such aspects of an object-oriented application such as the use of polymorphism, concurrency and dynamic binding.

8 Future Work

We are continually refining and extending the set of metrics defined in this paper.

There are plans to extend this work in a number of ways.

The next step will be to characterize a large number of benchmark and real world GUI applications using these metrics. It is planned to use the results from this to attempt to define the characteristics of GUI versus batch style execution scheme benchmark programs.

We also hope to illustrate the ability of such metrics to provide a measure of the object-orientedness of an application.

It is also planned to investigate the relationship (if any) that may exist between run-time measures of complexity and software quality.

It is also a goal to improve on the additional performance overhead that results from the use of the JPDA during the collection of the dynamic trace information.

It is important to recognise that the results presented in this paper are of a preliminary nature, and do not provide a justifiable basis for generalisation. However, we believe that they do provide an indication that the evaluation of software metrics at run-time can provide an interesting quantitative analysis of a program.

Acknowledgments

This work is funded by the Embark initiative, operated by the Irish Research Council for Science, Engineering and Technology (IRCSET).

References

[1] Abbot Java GUI Test Framework, Available at the following WWW site: `http://abbot.sourceforge.net`.

[2] Java Platform Debug Architecture (JPDA), Available at the following WWW site: `http://java.sun.com/products/jpda`.

[3] JUnit Testing Framework, P., Available at the following WWW site: `http://www.junit.org`.

[4] Memon, A.M., "GUI Testing: Pitfalls and Process", *IEEE Computer*, Vol. 35, no. 8, pp. 90-91, August 2002.

[5] Memon, A.M., Soffa, M.L. and Pollack, M.E., "Coverage Criteria for GUI Testing", *8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, Vienna University of Technology, Austria, Sept. 10-14, 2001.

[6] Mitchell, A. and Power, J.F., "Towards a definition of run-time object-oriented metrics" *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2003)*, Darmstadt, Germany, July 22, 2003.

[7] Ronsse, M., De Bosschere, K., Christiaens, M., Chassin De Kergommeaux, J. and Kranzlmuller, D., "Record/Replay for Nondeterministic Program Executions", *Communications*, Vol. 46, no. 9, pp. 62-67, Sep 2003.