

Toward a definition of run-time object-oriented metrics

- Position Paper -

Aine Mitchell, James F. Power

Abstract—This position paper outlines a programme of research based on the quantification of run-time elements of Java programs. In particular, we adapt two common object-oriented metrics for coupling and cohesion, so that they can be applied at run-time. We demonstrate some preliminary results of our analysis on programs from the SPEC JVM98 benchmark suite.

Index Terms—Area V (Metrics Validation): Formal and empirical validation of OO metrics, Standard data sets for metrics validation.

I. INTRODUCTION

Software metrics measure different aspects of software complexity and therefore play an important role in analysing and improving software quality. Measures of software complexity, for example metrics for coupling or cohesion, provide a means of quantifying its internal quality. Internal Quality measures are those which can be performed in terms of the software product itself and they will be measurable during and after the creation of the software product. However they have no inherent, practical meaning within themselves. To give meaning to these measures they have to be characterised in terms of the External Quality. External Quality measures are evaluated with respect to how a product relates to its environment. The reliability, maintainability, testability and reuse of a product are some examples. These measures are deemed to be inherently meaningful. Previous studies have indicated that software metrics can be used to obtain useful information on external quality aspects of software [1] [5].

Traditional metrics for measuring software such as Lines of Code (LOC) have been found to be inadequate for the analysis of object-oriented software

[11]. LOC was first proposed in the late 1960's and since then has been routinely used as a basis for measuring different notions of program size, for example programmer productivity (as LOC per programmer per month) and program quality (as defects per KLOC). However it was soon seen that using this alone as a measure for different notions of program size such as effort, functionality and complexity was insufficient. After all the number of Lines of Code in an assembly language is not comparable in effort, functionality, or complexity to the number of Lines of Code in a high level language, for example Java. With the growing diversity of different programming languages the need for a range of more discriminating measures became more apparent.

In recent years many researchers and practitioners have proposed a number of design metrics for object-oriented software, for example, the suite of metrics proposed by Chidamber and Kemerer [7] [8]. These design metrics quantify different aspects of the complexity of the source code, based on a static analysis of that code. Static metrics are defined as measures that quantify what *may* happen when a program is executed, as opposed to run-time metrics, which evaluate what *actually* happens. The ability of such static metrics to accurately predict the run-time complexity of an application is as yet unproven. The use of techniques such as multi-threading, recursion, and data driven algorithms in object-oriented languages make static analysis of the source code increasingly misleading. It is often not possible to predict in any way how a particular instance of code will operate until it is evaluated at runtime in a variety of contexts. For these reasons static metrics alone may be insufficient in evaluating the run-time complexity of an application, as its overall complexity will be influenced by the operational environment as well as the complexity of the

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

Please address correspondence to ainem@cs.may.ie

source code. Research by Yacoub et. al. [17] has indicated that useful information may be obtained from a measure of quantifying the dynamic complexity of software in its operational environment. However [17] only considered the use of such metrics in the design stage of the software lifecycle. A thorough literature survey showed that no previous study had endeavored to develop metrics that could be applied at run-time.

II. RELATED WORK

As object-oriented design techniques have become increasingly important a large number of object-oriented metrics for statically evaluating a design have been proposed. Coupling and Cohesion metrics are two such measures. They are said to evaluate the external and internal complexity of a design respectively. A large body of research has gone into investigating how these complexity measures characterise the external quality attributes of a design, for example its maintainability, reusability or error-proneness.

Our hypothesis is that it is possible to quantify the external quality of a software product using a set of run-time metrics that evaluate the product's complexity. These metrics by themselves can provide us with useful information, and can also help to calibrate the information obtained from a static analysis.

Briand et.al. [3] carried out an extensive survey of the current available coupling literature in object-oriented systems and concluded that all the current metrics measured coupling statically, at the class level. No measures of (dynamic) object level coupling had been proposed. They suggested that the reason for this is the obstacle of determining the degree of coupling or cohesion between individual objects. They proposed that a way of evaluating these would be to find some method of instrumenting the source code to log all occurrences of object instantiations, deletions, method invocations, and direct reference to attributes while the system is executing.

A study was conducted by Gupta and Rao [13] comparing a program execution based approach of measuring the levels of module cohesion present in legacy software, with a static-based method. The results from this study showed that the static-based approach significantly overestimated the levels of

cohesion present in the software tested, therefore indicating that a dynamic measurement would prove useful.

Yacoub et.al. [17] proposed a suite of dynamic metrics concerned with evaluating the quality of a design during the early development phase. This suite contained two metrics for determining coupling between objects, Import Object Coupling (IOC) and Export Object Coupling (EOC). These measures were obtained at an early development phase from executable object-oriented design models, which were used to model the application to be tested. They are both based on execution scenarios, that is "the measurements are calculated for parts of the design model that are activated during the execution of a specific scenario triggered by an input stimulus." The scenarios were then extended to have an application scope.

Despite the extensive research in this area no measures for quantifying coupling and cohesion at run-time have been proposed. The quality of a software product will be influenced by its operational environment as well as the source code complexity, therefore it was believed that measures that assess run-time quality may aid in the analysis of software quality. It is notable that such a study involves a change in the orientation of the metrics: static coupling and cohesion metrics deal with the *architectural* aspects of a software system, whereas run-time measures necessarily deal also with the *behavioural* aspects of the system.

For example, suppose a class is determined to be cohesive using the standard static metrics. This would imply that a static analysis of the programs has determined that the methods in the class access or change the full range data attributes. However, it is conceivable that at run-time, a statically cohesive class may not exhibit cohesive behaviour. That is, the methods that detract from cohesive behaviour might far outweigh those that contribute to cohesion when weighted by the number of times that they are invoked.

Similarly, the standard coupling metrics are typically defined as "coupling between objects (CBO)", yet they actually measure coupling between *classes*. A method that is judged by static metrics to contribute to coupling between two classes may in fact be rarely invoked, thus leading to a lower actual coupling between objects at run-time.

As a further example, some classes have objects

that exhibit a state-based behaviour. That is, we can distinguish various stages in the lifetime of their objects: perhaps an initialisation stage followed by a period of activity, followed by a finalisation phase. It seems reasonable that some objects may exhibit high degrees of cohesion and low coupling during initialisation, but exhibit low cohesion and high coupling during their active phase. Static metrics fail to identify or quantify this behaviour, whereas this information should be available from run-time metrics.

An important issue that must be addressed in relation to run-time metrics is dynamic binding. Static metrics are somewhat constrained in their ability to deal with inheritance issues [2] [10] since the run-time types at field access and method invocation sites are not known. However, run-time metrics *must* deal with such issues, and our initial work suggests that this can lead to a divergence with statically predicted results.

III. WORK DONE TO DATE

We have conducted a preliminary study on formulating definitions for coupling and cohesion at run-time. These run-time metrics were applied to assess the quality of a number of Java programs from the Java Grande Forum [6] and SPEC JVM98 [16] benchmark suites. The results from this preliminary analysis are available in two technical reports [14] [15].

We initially obtained a run-time profile of Java applications by instrumenting the Kaffe Virtual Machine to log all occurrences of object creation, method calls etc. More recently, we have switched to using Sun's JVM as the corresponding class libraries are more conformant, and it now has a built in profiler, all of which will make the analysis more easily repeatable.

At present we are utilising the Java Virtual Machine Debug Interface (JVMDI), a programming interface contained in the J2SDK from version 1.4.0 onward. JVMDI is one layer within the Java Platform Debugger Architecture. The second layer is the Java Debug Wire Protocol which defines the format of information and requests transferred between the process being debugged and the debugger front end, which implements the Java Debug Interface. The Java Debug Interface, which is the third layer, defines information and requests at the user code

level. The JDI provides introspective access to a running virtual machine's state, the class, array, interface, and primitive types, and instances of those types.

We are currently concentrating on programs written in the Java programming language, but our techniques should also apply to other object-oriented languages. Since all Java programs are executed on a virtual machine, this provides an ideal platform for their profiling and dynamic analysis. One of the problems with dynamic analysis that has been eluded from previous studies is the large volume of data that is generated. Thus the organisation and classification of this data will need to be given careful consideration.

Results obtained from this study indicated that valuable information on software quality might be obtainable from a run-time evaluation. It is known that a one-to-one relationship does not exist between static and dynamic metrics but our study indicated that there might be some co-relation. Hence there is a need for further empirical studies to validate these metrics and explore the dependency of design quality on each.

A. Relationship with Software Testing

The benchmark programs used to date may not be typical of Java applications. Indeed, it is notable that they vary widely both in the use of the Java class libraries [9] [12], and in their object creation profiles. Our next step will involve the profiling and analysis of more "real world" programs. In particular, it will be desirable to study object-intensive programs, such as GUI-based applications. Such programs however do not easily lend themselves to the batch-style running schemes of the standard benchmark suites. Thus, a strategy for choreographing and recording program runs will have to be developed.

Indeed, some of the issues associated with profiling benchmark suites can be seen in Table I. This table shows the number of objects created and methods called for the programs in the SPEC JVM98 benchmark suite. Each benchmark suite can be executed in one of three standard sizes $s1$, $s10$ or $s100$, with $s1$ being a simple calibration test. As can be seen in Table I, the behaviour of the applications as the length of the run increased is not always as expected.

TABLE I

PROGRAM SIZE DATA FOR PROGRAMS IN THE THE SPEC JVM98 BENCHMARK SUITE. EACH PROGRAM WAS RUN AT THREE SIZES: 1, 10 AND 100.

SPEC JVM98						
Application	Number of Objects created			Number of Method Calls		
	Size			Size		
	s1	s10	s100	s1	s10	s100
<i>_201_compress</i>	8,834	9,032	8,902	17,163,803	15,966,749	17,822,835
<i>_202_jess</i>	85,264	189,698	1,846,358	635,440	6,077,035	23,333,274
<i>_205_raytrace</i>	552,326	1,085,055	1,695,963	5,772,110	19,559,308	25,577,699
<i>_209_db</i>	13,520	176,459	3,554,259	136,726	2,299,380	34,183,241
<i>_213_javac</i>	64,081	382,060	1,553,455	443,790	3,319,026	14,050,905
<i>_222_mpegaudio</i>	15,824	18,035	15,215	1,185,653	9,368,543	21,452,712
<i>_227_mtrt</i>	551,114	1,567,812	1,955,785	5,758,391	23,772,645	25,856,949
<i>_228_jack</i>	484,952	953,582	4,009,120	4,007,716	7,921,292	33,064,049

Clearly, the first row of data indicates that *_201_compress* creates most of its objects and calls most of its methods during the initialisation phase of the benchmark. Extra iterations of the benchmark's inner loop, as represented by the increase in size from *s1* to *s100*, do not greatly affect either the number of objects created or the number of methods called. In the sixth row of data in Table I, the *_222_mpegaudio* program shows similar object creation patterns, but the number of method calls does increase for more iterations. Finally, most of the other programs, for example *_202_jess* on the second row, display an increase in both object creation and methods called, albeit at different rates.

Thus, along with the metrics, we hope to define a methodology for quantifying real-world applications with these metrics. We would hope to exploit some of the existing work on software testing in this area. We also hope to determine if a quantifiable correlation exists between the information obtained from a static and dynamic analysis of a program. Clearly, a static analysis is relatively independent of program behaviour, whereas any run-time analysis will be fundamentally influenced by the testing strategy and test inputs. This issue must be addressed in any assessment of run-time metrics, and may have implications for the quantification of the effectiveness of software testing strategies.

IV. SOME PRELIMINARY RESULTS

In this section we present some preliminary results from our analysis of the SPEC JVM98 suite. 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.

A. Coupling Metrics

The Coupling between Objects (CBO) measure was originally defined as "a count of the number of non-inheritance related couples with other classes". Two objects are deemed to be coupled if they *act upon one another*, in other words, if an object of one class uses the methods or instance variables of the other [7]. If a method declared in one class uses a method or instance variable in another class, this pair of classes are said to be coupled since all objects instantiated from the same class are deemed to have the same properties.

However Chidamber and Kemerer later revised their definition of CBO. For a class C, CBO is a measure of the number of other classes to which it is coupled [8]. We can extend this directly to define the **Dynamic CBO for a class** as being a count of the number of couples with other classes at run-time.

For a whole program, we can define the **Degree of Dynamic Coupling** within a given set of classes as follows:

$$\frac{\text{Sum of number of accesses to methods or instance variables outside each class}}{\text{Sum of total no. of accesses from these classes}} * \frac{100}{1}$$

Differences between the static and dynamic values for this metric result from the relative frequency

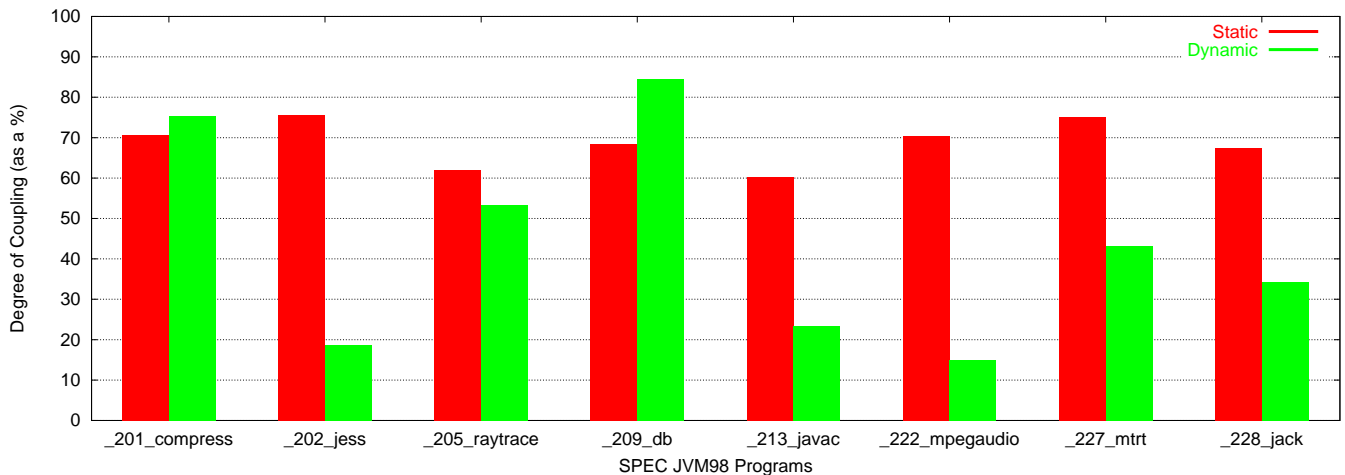


Fig. 1. Degree of Static and Dynamic Coupling within a group of classes for the non-API classes of the programs (Size s100) from SPEC JVM98 benchmark suite.

with which methods that contribute to coupling are called. Figure 1 summarises the results for the programs in the SPEC JVM98 suite. As can be seen from Figure 1, the static values may be either less than or greater than the dynamic values, depending on the program being run and, presumably, on the input data supplied to it. One of the goals of our research is to provide a detailed explanation for this behaviour.

B. Cohesion Metrics

Chidamber and Kemerer [8] defined a static cohesion metric for object-oriented applications known as Lack of Cohesion in Methods (LCOM). Suppose a class contains n methods, m_1, \dots, m_n . Then Let $\{I_i\}$ be the set of instance variables referenced by method m_i . We can define two sets:

$$P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$$

$$Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$$

The (static) LCOM is then equaled to $|P| - |Q|$, if $|P| > |Q|$, and 0 otherwise.

Figures 2 and 3 give a pictorial representation of the relationship between the static and dynamic cohesion data for the SPEC JVM98 benchmark program *_209_db*. Here, each class is represented by a graph whose nodes represent the methods of that class. For any two methods m_i and m_j , there is an edge between m_i and m_j precisely when the intersection between the set of instance variables accessed by these methods is non-empty. Comparing the results for the static analysis of Figure 2 and the

dynamic analysis of Figure 3, it can be seen that the static results may considerably overestimate those produced by the dynamic analysis.

In Table II we compare the static and dynamic LCOM values for the *Database* class of the *_209_db* program. We define the **Simple Dynamic LCOM** analogously to the static version, but only counting instance variables that are actually accessed at run-time. Thus, these values will always be less than their static counterpart. As can be seen from the second column of data in Table II this decrease is not uniform between measures P and Q , and the *Database* class demonstrates a lack of cohesion at run-time.

We can define **Dynamic Call-Weighted LCOM** by weighting each instance variable by the number of times it is accessed at run-time. The third column of data in Table II presents the corresponding values of P and Q for the weighted case. Here, the methods contributing to the lack of cohesion are accessed less than the cohesive ones, thus suggesting that the class might be viewed as cohesive after all.

Clearly, the simple static calculation of LCOM as $|P| - |Q|$ masks a considerable amount of the detail available at run-time.

V. CURRENT AND FUTURE WORK

We plan to proceed with our research in three main stages:

- **Stage I:** Starting with the standard statically calculated object-oriented metrics for coupling and cohesion [3] [4], we are presently looking

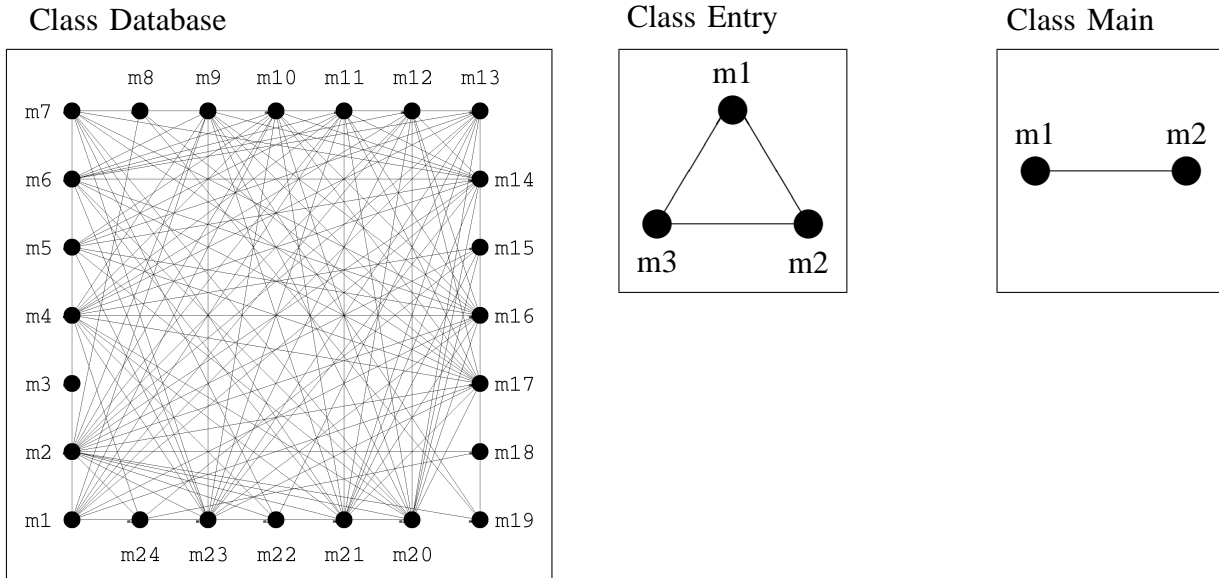


Fig. 2. Diagrammatic Representation of Static LCOM for Non-API Classes involved in execution of *_209.db* program (Size s100) from the SPEC JVM98 benchmark suite.

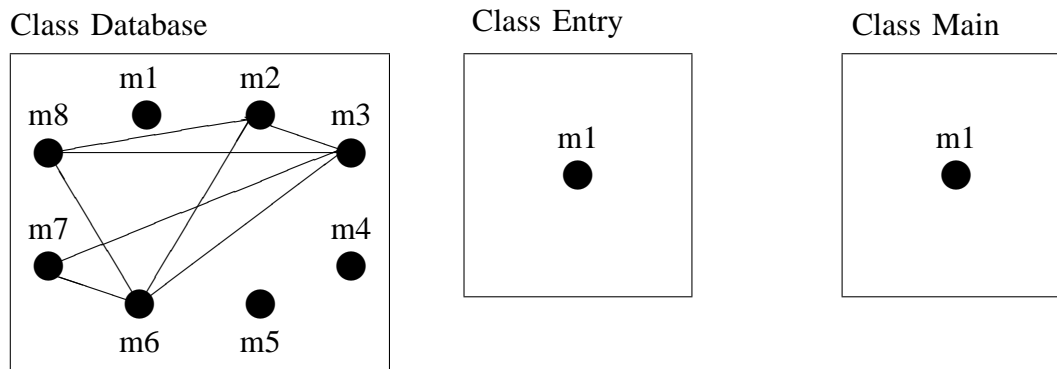


Fig. 3. Diagrammatic Representation of Dynamic LCOM for Non-API Classes in execution of the *_209.db* program (Size s100) from the SPEC JVM98 benchmark suite.

at the options for defining run-time versions. We have already specified a number of these, and hope to provide a broadly-based definition and categorisation of the possibilities here.

- **Stage II:** Our present work has mainly involved benchmark suites such as SPEC and Grande. We propose to widen this collection of programs to include more common “real-world” Java applications. There are various technical problems to be solved here in terms of running the programs in a documented, repeatable manner, and in generating and processing the profiling information.

- **Stage III:** Run-Time metrics can only be justified if they provide additional useful information either about the programs themselves, or about the test cases used for the program’s run. A key aspect of our study will be the analysis and comparison of the various possible definitions of run-time metrics in order to determine their possible utility.

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

TABLE II

STATIC AND DYNAMIC LCOM VALUES FOR THE *_209.db*
PROGRAM (SIZE S100) FROM THE SPEC JVM98 BENCHMARK
SUITE.

	Static	Simple Dynamic	Call-Weighted Dynamic
$ P $	120	20	95,394
$ Q $	156	8	125,330
LCOM	0	12	0

interesting quantitative analysis of a program.

REFERENCES

- [1] Basili, V.R., Briand, L.C. and Melo W.L., "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, Vol. 22, no. 10, pp. 751–761, October 1996.
- [2] Bieman, J.M. and Kang, B.K., "Cohesion and Reuse in an Object-Oriented System," *Proc. ACM Symp. Software Reusability (SSR'94)*, pp. 295–262, 1995.
- [3] Briand, L.C., Daly, J.W. and Wust, J.K., "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.: An Int'l J.*, Vol. 3, no. 1 pp. 65–117, 1998.
- [4] Briand, L.C., Daly, J.W. and Wust, J.K., "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Eng.*, Vol. 25, no. 1 pp. 91–121, Jan/Feb 1999.
- [5] Briand, L.C., "Empirical Investigations of Quality Factors in Object-Oriented Software," *Empirical Studies of Software Engineering*, Ottawa, Canada, March 4–5, 1999.
- [6] Bull, J. M., Smith, L.A., Westhead, M.D., Henty, D.S. and Davey, R.A., "Benchmarking Java Grande Applications", in *Proceedings of The Second International Conference on The Practical Applications of Java*, Manchester, U.K., April. 2000, pp. 63-73.
- [7] Chidamber, S.R. and Kemerer, C.F., "Towards a Metrics Suite for Object-Oriented Design," *Proc. Conference on Object-Oriented Programming: Systems, Languages and Applications*, (OOPSLA'91), SIGPLAN Notices, Vol. 26, no. 11, pp. 197–211, 1991.
- [8] Chidamber, S.R. and Kemerer, C.F., "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, no. 6, pp. 467–493, June 1994.
- [9] Daly, C., Horgan, J., Power, J. and Waldron, J. "Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite", *Joint ACM Java Grande - ISCOPE 2001 Conference*, Stanford University, USA, 2-4 June, 2001.
- [10] Eder J., Kappel G. and Schrefl M. "Coupling and Cohesion in Object-Oriented Systems" *Technical Report*, University of Klagenfurt, 1994.
- [11] Fenton, N.E. and Neil M., "Software Metrics: Successes Failures and New Directions," *The Journal of Systems and Software*, Vol. 47, pp. 149–157, 1999.
- [12] Gregg, D., Power, J.F. and Waldron, J. "Benchmarking the Java Virtual Architecture - The SPEC JVM98 Benchmark Suite", Chapter 1 of *Java Microarchitectures*, Ed. N. Vijaykrishnan and M. Wolczko, Kluwer Academic, 2002.
- [13] Gupta, N. and Rao, P. "Program Execution Based Module Cohesion Measurement," *16th International Conference on Automated on Software Engineering (ASE '01)*, San Diego, USA, November 2001.
- [14] Mitchell, A. and Power, J.F., "Run-time Coupling Metrics for the Analysis of Java Programs - preliminary results from the SPEC and Grande suites" Technical Report NUIM-CS-TR2003-07, Dept. of Computer Science, NUI Maynooth, Ireland, April 2003.
- [15] Mitchell, A. and Power, J.F., "Run-time Cohesion Metrics for the Analysis of Java Programs - preliminary results from the SPEC and Grande suites" Technical Report NUIM-CS-TR2003-08, Dept. of Computer Science, NUI Maynooth, Ireland, April 2003.
- [16] Standard Performance Evaluation Corporation, SPEC JVM98 Benchmarks, Available at the following WWW site: <http://www.spec.org/jvm98>.
- [17] Yacoub, S.M., Ammar, H.H. and Robinson, T., "Dynamic Metrics for Object-Oriented Designs," *5th International Software Metrics Symposium*, Boca Raton, Florida, USA, pp. 50–61, Nov 4–6, 1999.