An Extensible Metrics Extraction Environment for Object-oriented Programming Languages

T.J.Harmer
School of Computer Science
The Queen's University of Belfast
Belfast
BT7 1NN
Northern Ireland
(email: t.harmer@qub.ac.uk)

F.G.Wilkie
Centre for Software Process Technologies
Faculty of Informatics
The University of Ulster
Newtownabbey
Co-Antrim
BT37 0QB
Northern Ireland
(email: fg.wilkie@ulster.ac.uk)

Abstract

In this paper the enhancements to an existing production-quality compiler to facilitate the collection of complexity metrics as an integral part of the compilation and build process are described. An object oriented programming language meta-model is presented. This meta-model is used in a relational database schema and enables a language neutral form of a compilation unit of source code or system of units to be stored. We have used C language routines with embedded SQL to calculate a suite of 11 complexity metrics for each class stored in the database. We present the full implementation of one of these routines. The resulting metrics extraction tool is highly extensible, readily facilitating incorporation of new object oriented programming languages and new metrics.

Keywords – Programming language Metrics, Object-oriented metrics, Complexity, C++, Java.

1. Introduction

A software complexity metric is intended to provide an indicator of the quality of a software system design or a software system implementation. For example, a metric might attempt to indicate the degree of fault-proneness [1], [2], the impact of software changes in software maintenance [3], [4], [5] or the ease of change in program evolution [6].

Many complexity measures for object-oriented systems exist in published literature. One very popular source is that by Chidamber and Kemerer [7], [8] who proposed a suite of six metrics to measure factors such as class size; class coupling by virtue of inheritance; class coupling by virtue of other linkages primarily used during message passing interaction; and class cohesion. Other metrics have been proposed such as the Message Passing Coupling metric and the Data Abstraction Coupling metric [9]; the Information Flow metrics [10]; the Coupling Factor metric [11]; the Method-Method interaction measures [12]; Coupling Complexity [2] and extensions to the Chidamber & Kemerer cohesion metric (LCOM) proposed by Hitz and Montazeri [13][1], to name only a few.

The myriad variables in the software development process makes interpretation of metrics difficult; for example, the value of a software metric will be influenced by the type of application domain, the choice of programming language, the skill level of development staff, and the team and the client management approach. Beyond a recognition that *coupling*, the degree of interconnectedness of system components, and *cohesion*, the strength of elemental relationships within components, are of fundamental importance, there is little agreement that any of the existing metrics are particularly useful.

If software metrics are to be moved from being a research interest to a serious software development support tool then empirical studies of the utility of metrics as indicators of software quality are required. Such empirical studies should consider metrics applied to practical, commercial software development projects and

[1] As discussed in [14].

track the evolution of the product and the metrics. Thus the relationships between metrics and the observed qualities of a software product can be related. (We are engaged in such a long-term study as discussed in [15].

The large-scale nature of commercial software development requires an automated means of metric data collection; indeed, the manual collection of metric data is only feasible for small software systems. There are many automated tools available for performing metrics analysis (for example [16] and [17]). Most of these tools can only be considered research prototypes since the usefulness of their results is still a matter of debate.

Given this problem, we report on the design and development of a metrics analyser designed for use with large-scale software development projects. The tool, called EMBER (**E**xtensible **M**etrics tool**B**ench for **E**mpirical **R**esearch), is designed to be extensible, to handle multiple computer programming languages and be easily integrated within a software development environment. We believe that with the current state of research, it is essential that any tool used to gather metrics data is easily modified and extended to accommodate both new and evolving metrics, and permit specialised metrics of use in specific application areas to be developed quickly. The capability to make measures of software systems developed in multiple programming languages reflects our interest in practical software development projects, which are often multi-language. Both system level and component level measures are important.

Through consistency, our approach enables a better comparison of metrics data for software components implemented across a range of object oriented programming languages. Integrating metric data collection with conventional software development is an attempt to move metric collection from a research or review tool to a practical software developer aid. If metrics are meaningful then it is reasonable to permit software developers to view metric results as part of normal software development processes.

2. EMBER Metric Architecture

The basic requirements of the EMBER metric architecture are

- *extensibility;*

  The architecture attempts to make it possible to develop new metrics easily with little (and hopefully no) impact on other parts of the architecture

- *multi-language capability;*

  The architecture records data in a language neutral form and permits metrics to be calculated without knowledge of the native language in use. This permits metrics to be calculated in a consistent fashion on a multi-language software system implementation.

- *integration within the software development environment*.

  The architecture should fit naturally within a software development environment enabling metrics to be calculated as software components are developed and integrated.

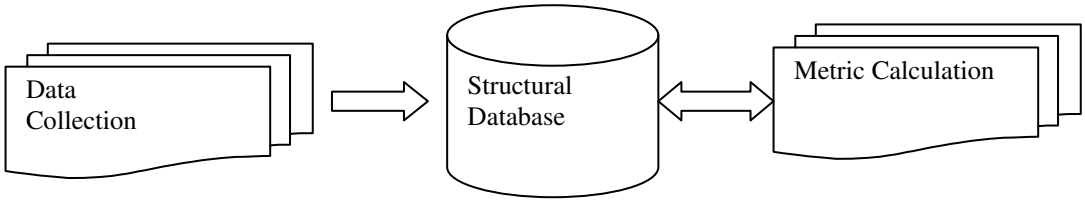

Figure 1. Ember Architecture

The architecture of EMBER, depicted in Figure 1, separates data collection from metric calculation. Data detailing the structure of a program is collected as each software component is compiled and stored in a shared database. Individual metric calculation tasks to derive metrics for the system as a whole, or distinct system components use this shared structural database. The separation of the data collection tasks from metric calculation tasks makes it possible to collect data for each programming component separately. This facilitates metric calculation for multi-language software systems and thus makes the EMBER architecture highly extensible because distinct, language-specific data collection modules can be developed with each module storing data in a shared, language-neutral form. The collection of structural data

while a software component is being compiled permits integration within a software development process and couples metric analysis with a software development process. Indeed, metrics can be calculated for single components or subsets of components and so the tool integrates naturally into individual, team, and group software development processes.

2.1. A Language Meta-Model

A programming language specific parser collects data on program structure as a software component is compiled and stores this data in a shared database. This process maps a language-specific format, a program, into a language-neutral format that is recorded as a database schema. The development of a meta-model for the database schema maximises the ability of the tool to accommodate other object oriented programming languages. Therefore, to convert from analysing C++ to Java only requires changing the parser. The database schema remains unchanged. Likewise, as the database de-couples the metric data collection from metrics calculation, analysis of the language-neutral form, as present in the database, is independent[2] of the original target language. The database schema for the language neutral form is illustrated across figures 2 and 3 for presentational purposes.

Figure 2 depicts the database entities (rectangles) and structural relationships (arcs) between these entities used by the EMBER architecture. The entities record the elements that are involved in metric calculation: *Data* that record values in a computation; *Method*s that group program statements; *Class*es that define objects in an object oriented program; *Package*s that structure an object oriented system; and *File*s that store the program text.

The relationships between entities define the software system structure:

- Data may be defined within a File (is_file_data_of), or a Package (is_package_data_of), or a Class (is_class_data_of), or a Method (is_method_data_of or is_argument_of);
- Methods may be grouped within a File (is_file_method_of), or a Package (is_package_method_of), or a Class (is_class_method_of), or a Method (contains);
- Classes may be grouped within a Package (is_grouped_under), or a Class (is_class_of) ;
- Classes may have a collection of parent Classes (is_parent_of) and a collection of ancestor Classes (is_ancestor_of); and the Class text is stored in a File (is_declared_in); and
- Packages may be grouped within a Package (is_part_of).

These structural relationships permit the calculation of metrics such as: Weighted Methods per Class (WMC), Coupling between Object Classes (CBO), Depth in the Inheritance Tree (DIT) and Number of Children (NOC) – from[8].

The relationships, depicted in Figure 3, record system dynamic behaviour:

- *Data* may be changed within:
  *File*s (*file_data_access*), or
  *Package*s (*package_data_access*), or
  *Class*es (*class_data_access*), or
  *Method*s (*method_data_access*);

- *Method*s may be invoked within:
  *File*s (*file_method_invocation*), or
  *Package*s (*package_method_invocation*), or
  *Class*es (*class_method_invocation*), or
  *Method*s (*method_invokes*).

These relationships permit the calculation of metrics such as: the Response Set for a Class (RFC) and the Lack of Cohesion amongst Methods (LCOM) metrics [8], as well as the forwards and backwards Coupling Complexity metrics (CCF and CCB) [2] and the Message Pasing Coupling Metric (MPC) [9].

In designing the database schema, both C++ and Java languages were considered. Java has a 'package' concept that does not exist in C++. Therefore, as far as C++ is concerned, the resulting schema has some redundancy. From a brief consideration of object versions of Pascal (viz Borland's DELPHI product) we believe that the schema is sufficient for their needs as well. (The analysis of systems composed of C, C++ and Java components is our current research interest.)

The database schema has been implemented using the MySQL relational database package (see www.mysql.com). MySQL provides an efficient and fast access data storage capability and can cope with a high volume of data. The schema has also been prototyped in other available database packages, such as PostgreSQL (see www.postgresql.org).

[2] In as much as the metric being measured is applicable to range of object-oriented programming languages.

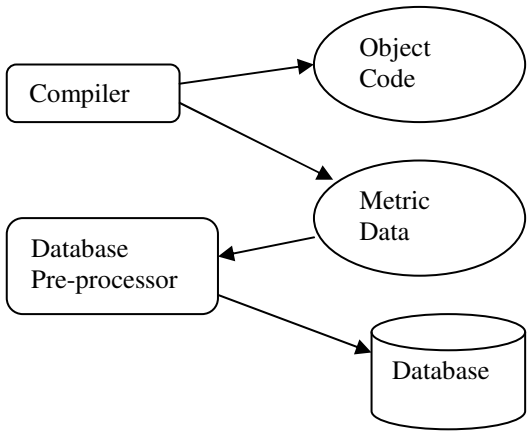

Figure 4. Metric data collection

2.2. Metric Data Collection

The collection of metric data from system components is performed as each component is compiled, as depicted in Figure 4. This enables the metric collection process to be coupled with the system development process and thus makes it possible to calculate metrics early in software development. However, software development can be a time of rapid change in component structure that would lead to rapid changes in a single, shared metric database. The EMBER system architecture permits a collection of metric databases to be used to model developer, team and group work. Thus a single developer can create a metric database for their component and obtain metrics appropriate to that component; a development team can create a team database from the stable team components and create team metrics etc.

The metric data produced by the language compiler module is recorded as a collection of simple text definitions, such as:

```
method(name = 'test', parent = 'testclass',
      visibility = 'public', mode = 'instance',
      type = 'void')
```

A metric data definition file for a component is self-contained and includes definitions of all external classes, data, libraries etc and thus enables a metrics database to be constructed for that component.

This text file, containing structural data, is read by a separate database component that creates the database schema outlined above. The database pre-processor separates metric collection from database construction and permits multiple database implementations to be used. The pre-processor also performs utility functions such as generating unique identification numbers for each entity (to enable non-text based identification) and construction of relationships such as defining the collection of class ancestors.

Metrics data is produced automatically as a software component is compiled. The development of a compiler, indeed group of compilers for the target languages, requires significant effort; an effort certainly beyond the scope of our work. However, it is possible to gather metrics information by extending the freely available GNU compiler suite (see www.gnu.org). Further, the GNU compiler suite outputs much of the data required for metrics calculation, if the appropriate compiler options are selected.

The GNU compiler suite permits the output of debugging data that permits source-level program debugging. For example, variable names, method names and type information are output to permit a user to inspect data and track program progress as a program is run within a debugging environment. The debugging data is used by tools such as gdb (see www.gnu.org) and is stored as part of the object code file for a program component. This debugging data is the basic information that is also required for metrics calculation.

As part of the EMBER architecture, the GNU compiler suite was extended to format such debugging data and store it in a separate metric data file (as depicted in Figure 4). Almost all of the structural data required by entities and relations depicted in Figure 2 is output as part of debugging data. Minor extensions were necessary to output additional data required for metric calculation, such as classes defined by a component, class within which a method/variable is defined and the inheritance hierarchy. This additional data is not required for debugging purposes but is readily available within the compiler.

The extensions to the GNU compiler suite are self-contained and do not impact the normal operation of the GNU compiler(s). The metric data output can be switched on or off by setting an appropriate compilation flag and the writing of debugging data to the object code file is unaffected. The extensions to the Java, C and C++ compilers are the same in each case.

In Figure 3 additional relations that record information about the dynamic behaviour of a program are depicted. These are necessary for the calculation of certain metrics discussed previously. The data required to establish these relations is not output as part of debugging data from the GNU compiler suite. It would be possible to extend the compiler to output such data (as the compiler text is freely available). For example, where the compiler must process a call to a method, this routine might be extended to record the details for metric calculation purposes (and thus make it possible to establish the relation *method_invokes*). The data required to form the relations depicted in Figure 3 is

significant and would require many changes in the compiler text. We considered this approach too invasive. It would require similar changes in other languages being processed, making the support of multiple languages time consuming.

The GNU compiler suite uses a common intermediate language prior to object code generation. A language compiler translates a program source text to this intermediate format and a machine-specific code generation module translates from this format to create an object code file. This has the advantage that the machine-independent, language recognition part of compilation is separated from the machine-specific, object code generation part of compilation. This language-independent form is readily available and can be processed to extract the dynamic information required by metric calculation. Indeed, the compiler provides an exemplar routine that formats and prints this intermediate definition as part of compiler debugging activities.

By extending this exemplar routine we were able to collect information about method calls, data accesses, operator usage and statement structure necessary for the dynamic relations in our database schema. Furthermore, we can do this in a (largely) language-independent manner that requires only a single line change in particular GNU compilers - a method call to the metric output routine. As with the structural data, the dynamic data is recorded as simple text definitions, such as:

```
call(from='source_func',
     target='target_func',
     line=34',file='example.cpp')
```

3. Calculating Metrics using SQL

The relational database management system that was chosen for the EMBER tool was MYSQL. It was chosen because of its rich suite of features, readily available documentation and freely available runtime executive.

To date, SQL code has been developed to calculate 11 complexity metrics. Obviously some metrics are more difficult to deduce than others. The implementation of each metric required the development of C source code with embedded SQL statements. Table 1 gives the number of lines of code involved in calculating each of the 6 basic Chidamber & Kemerer metrics [8]:

Table 1. Approximate size for the implementations of each of the C&K metrics.

C & K Metric	LOC
WMC	62 LOC - 4 distinct query types.
DIT	76 LOC - 4 distinct query types.
NOC	21 LOC - 2 distinct query types.
CBO	59 LOC - 6 distinct query types.
RFC	96 LOC - 7 distinct query types.
LCOM	82 LOC - 5 distinct query types.

Each metric required approximately 4 hours to implement. As an example, the full implementation of the Coupling Between Object classes (CBO) metric [8] is provided in appendix A which shows the 6 queries used in its calculation. If we were only interested in calculating CBO, then 6 distinct queries would not be necessary. The whole calculation could be compressed into 3 queries. However, we are calculating all six C&K metrics here and most of them share portions of data. For example, calculating the C&K Weighted Methods per Class (WMC) metric requires us to build a list of methods in each class. This uses 2 of the same queries used in calculating the CBO metric – namely:

"SQL_select_class_list" and
"SQL_build_methods_table".

By decomposing the queries in this manner, we can store interim results required in subsequent metrics calculations, thereby improving the overall efficiency of the calculation process.

In addition, we have implemented the Coupling Complexity metrics CCF and CCB proposed by Wilkie & Hylands [2]. The CCF metric, in particular, is difficult to calculate because it requires knowledge of all message passing between classes of objects in the application. The CCF metric required just 79 lines of code involving 6 distinct query types, while for CCB, 43 lines of code involving 3 distinct query types were required. Again, both CCF and CCB were implemented in about 4 hours each.

4. Related Work

Moser [18] has researched the area of meta-models for object oriented systems. He indicates that metamodels have been used to define software measures such as those of [8]. Indeed, he suggests that for control

of software development projects, managers "*may indirectly use metamodels via software measures that may only properly be defined via metamodels (whether documented or not).*". LaLonde & Pugh [19] have proposed the use of the in-built metalevel facilities provided in Smalltalk for the gathering of static metric information in systems constructed in Smalltalk.

We believe that our database schema presented in figures 2 and 3 is consistent with the metamodel of an object-oriented system description presented in Moser (1995).

Tichelaar & Demeyer [20] have developed a language independent exchange model for the FAMOOS Esprit project [21]. This project involved several re-engineering tools and so the driving force was the need to enhance interoperability between tools. They needed to parse several object oriented programming languages (C++, Java, Ada and Smalltalk) and provide the output in a common format that could be supplied to various tools performing a variety of functions such as visualization, metrics calculation, and structural reorganization.

The approach of [20] was to develop a core data model called FAMIX [22] for storage of parsed source code. This core model was then enhanced with a set of plug-in modules (e.g. [23], [24]) for each programming language, thereby accommodating the variation that exists between C++, Java, Ada and Smalltalk.

The FAMIX core model is, not surprisingly, similar to our model depicted in figures 2 and 3. However in FAMIX, scoping of classes, functions and variables via packages, is handled through a naming convention "package::subpackage::classname". Likewise, the naming of variables involves pre-pending the associated package, subpackage, class and method names. In our approach, additional relationships between the Class entity and the Data entity provide this information in a more manageable form for subsequent metrics processing.

The language plug-ins for FAMIX handle specific language features. For C++, the plug-in extends the core model to handle (amongst others) friends, abstract classes and virtual functions. Our model handles these features as attribute values within the Class, Data and Method entities as well as entities generated to support the entity relationships in the implemented model. The FAMIX C++ plug-in creates a new entity called SourceFile that models a file of source code. Our model directly incorporates just such an entity (called File in figures 2 and 3). Likewise, for Java, a FAMIX extension for Package is included. Again, a Package entity with the same semantics exists in our model.

The advantage of the FAMIX approach is that the core model is enhanced with specific language features as required whereas with our approach, the model is essentially a super-set of what would be required to support any particular object-oriented programming language and therefore contains redundancy.

5. Conclusions and Future Work

The use of a modified compiler provides a relatively seamless integration of the metrics collection process with normal development/compilation activities.

De-coupling the parsing from the metrics calculation has made the overall metrics extraction architecture highly extensible. It is capable of having new object oriented programming languages incorporated with relative ease. At the same time, ensuring consistency in the calculation of metrics across languages.

We have successfully harnessed a relational database and SQL to calculate a range of complexity metrics for object oriented software systems. In most cases, each new metric was implemented in about 4 hours, facilitating the extension of the metrics analysis programme in a short timeframe and thereby supporting the research process. Briand et al [12] provide a comprehensive listing of published complexity metrics. We believe that our language neutral form is sufficiently rich to enable calculation of all these metrics.

The GNU compiler suite is a production quality compilation environment that provides a good base for metric data collection. If an organization is using the GNU compilation suite then the extended GNU compiler will fit seamlessly within the development environment. However, if an organization is using another compiler suite it is possible to use the metric compiler as a stand-alone tool that is executed when a metric calculation is performed.

The ability to collect metrics as an integral part of the compilation process is, we believe, a very powerful facility that encourages the use of complexity metrics during testing stages of the software development lifecycle.

Our intention is to convert the SQL routines (see, for example, appendix A) to adhere to the Java Database Connectivity standard (JDBC) thereby further enhancing the usefulness of the tool in commercial development environments. In addition, we are currently converting the simple textual definitions used to record metric data into a collection of XML definitions.

The output from the EMBER tool is a spreadsheet containing a set of metrics values for each class in the analyzed application. Our research is primarily concerned with evaluating the predictive properties of these and other metrics that can easily be deduced from the underlying database. Can these metrics predict fault-

proneness, or other maintenance difficulties such as program understanding? Can these metrics identify those classes that require most unit testing, or careful integration and/or system testing? How can the myriad of values be aggregated effectively and presented in a suitable manner for use in a commercial software development environment?

6. References

[1] Basili, V.R. Briand, L.C. and Walcélio, L.M. 1996. "A Validation of Object-Oriented Design Metrics as Quality Indicators." IEEE Transactions on Software Engineering, 22(10), pp.751-761.

[2] Wilkie, F.G and Hylands, B. 1998. "Measuring Complexity in C++ Application Software". Software Practice and Experience, Vol. 28(5), pp513-546.

[3] Briand, L.C. Wüst, J. and Lounis, H. 1999. "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems." Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, Technical Report number ISERN-99-03.

[4] Wilkie, F.G. and Kitchenham, B.A. 2000. "Coupling Measures and Change Ripples in C++ Application Software." Journal of Systems and Software, 52, pp.157-164.

[5] Wilkie, F.G. and Kitchenham, B.A. 2001. "An Investigation of Coupling, Reuse and Maintenance in a Commercial C++ Application." Information and Software Technology, vol.43, pp801-812.

[6] Arisholm, E. 2001. "Empirical Assessment of Changeability in Object-Oriented Software". Thesis submitted for the degree of Dr. Scient., Dept. of Informatics, Faculty of Mathemetics and Natural Sciences, University of Oslo, Norway.

[7] Chidamber, S.R. and Kemerer, C.F. 1991. "Towards a metrics suite for object-oriented design". in Proc. OOPSLA '91, Sigplan Notices, 26(11), pp197-211.

[8] Chidamber, S.R. and Kemerer, C.F. 1994. "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, 20(6), pp476-493.

[9] Li, W. and Henry, S. 1993. "Object oriented metrics that predict maintainability". J. Systems Software, 23, pp111-122.

[10] Lee, Y.S. Liang, B.S. Wu, S.F. and Wang, F.J. 1995. "Measuring the Coupling and Cohesion of an Object Oriented Program Based on Information Flow". Proc. International Conference on Software Quality, Maribor, Slovenia.

[11] Abreu, F. Goulão, M. and Esteves, R. 1995. "Toward the Design Quality Evaluation of Object-Oriented Software Systems." Proc. 5th International Conference on Software Quality, Austin, Texas, USA.

[12] Briand, L.C. Daly, J.W. and Wüst, J. 1996. "A Unified Framework for Coupling Measurement in Object-Oriented Systems." Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, Technical Report number ISERN-96-14.

[13] Hitz, M. and Montazeri, B. 1995." Measuring Coupling and Cohesion in Object-oriented Systems." Proc. International Symposium on Applied Corporate Computing, Monterrey, Mexico.

[14] Briand, L.C. Daly, J.W. and Wüst, J. 1997. "A Unified Framework for Cohesion Measurement in Object-Oriented Systems." Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, Technical Report number ISERN-97-05.

[15] Wilkie, F.G. and Harmer, T.J. "Tool Support for Measuring Complexity in Heterogeneous Object-Oriented Software". Submitted to ICSM 2002, pending peer review.

[16] Stiglic, B Heričko, M. and Rozman, I. 1995. "How to evaluate Object-Oriented Software Development?" ACM SIGPLAN Notices, 30 (5), pp.3-10.

[17] Aggarwal, K.K. and Singh, Y. 1997. "A Tool for the Collection of Industrial Software Metrics Data". ACM SIGSOFT, Software Engineering Notes 22 (2), pp.54-57.

[18] Moser, S. 1995. "Metamodels for Object-Oriented Systems: A Proposition of Metamodels Describing Object-Oriented Systems at Consecutive Levels of Abstraction" Software - Concepts and Tools, 16, pp.63-80.

[19] LaLonde, W.R and Pugh, J. 1994. "Gathering metric information using metalevel facilities." Journal of Object oriented Programming, March-April 1994, pp.33-37.

[20] Tichelaar, S and Demeyer, S. "An Exchange Model for Reengineering Tools", ECOOP '98, LNCS 1543, Springer-Verlag, July 1998. (Editors: S.Demeyer and J.Bosch).

[21] FAMOOS Project, Esprit IV programme project no. 21975. http://www.iam.unibe.ch/~famoos/.

[22] Demeyer, S Tichelaar, S and Steyaert, P. "The FAMOOS Information Exchange Model", Technical Report, University of Berne, August 1999. http://www.iam.unibe.ch/~famoos/FAMIX/.

[23] Bär, H. "FAMIX C++ language plug-in 1.0", Technical Report, University of Berne, September 1999.

[24] Tichelaar, S. "FAMIX Java language plug-in 1.0", Technical Report, University of Berne, September 1999.

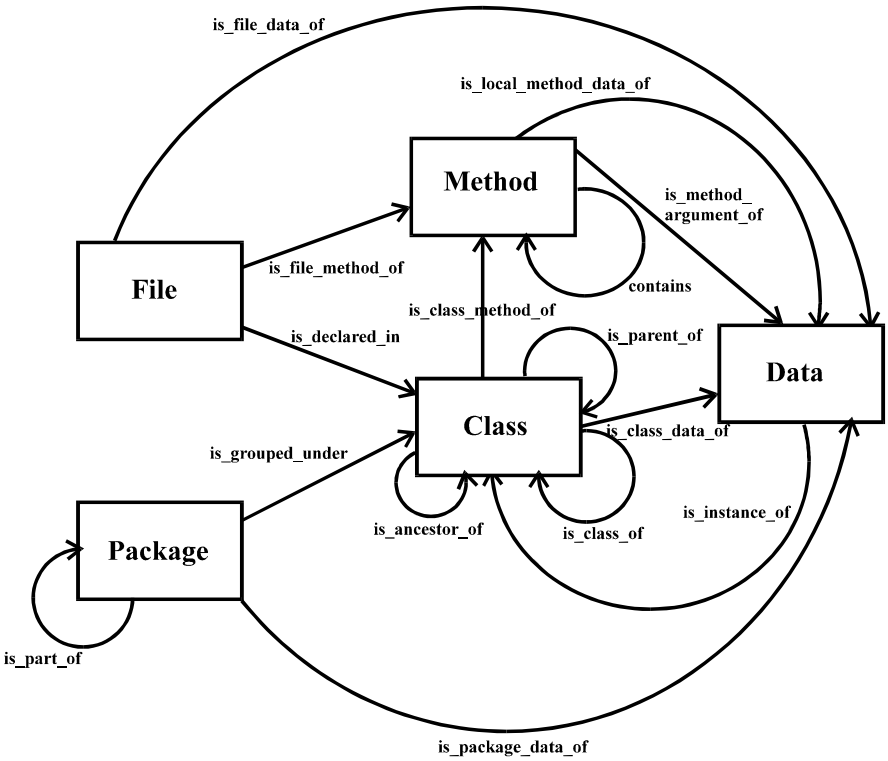

Figure 2. EMBER database structure

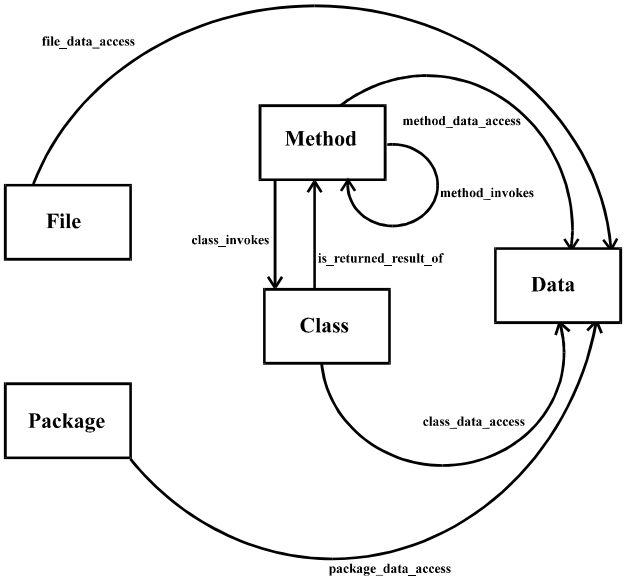

Figure 3. EMBER Dynamic behaviour database structure

Appendix A - SQL to build the CBO metric

```
// ****************************************************************************
// Functions created:
//                                                      1. calculate_CBO
//
// SQL queries produced:
//                                                      1. SQL_select_class_list
//                                                      2. SQL_build_methods_table
//                                                      3. SQL_build_methods_data_table
//                                                      4. SQL_build_all_data_table
//                                                      5. SQL_build_class_links_table
//                                                      6. SQL_select_class_links_table
//
// ****************************************************************************

// Embedded SQL to build C&K CBO metric
// This code assumes the 'conn' variable to be set previously

#include <string.h>
#include <stdio.h>
#include <mysql.h>

void SQL_select_class_list(char *) ;
void SQL_build_methods_table(char *, char *);
void SQL_build_methods_data_table(char *, char *);
void SQL_build_all_data_table(char *, char *);
void SQL_build_class_links_table(char *, char *) ;
void SQL_select_class_links_table(char *) ;

// ----------------------------------------------------------------------------

void calculate_CBO(MYSQL *conn)
        {
        // Calculates Chidamber & Kemerer CBO Metric for all classes in the list

        MYSQL_RES *class_results_set, *linked_class_results_set;
        MYSQL_ROW class_row, linked_class_row;
        int cbo ;
        char query[1024];

        // First of all we need to determine the list of classes for which
        // CBO will be calculated.

        SQL_select_class_list(query);

        handle_mysql_query(conn, query);

        class_results_set = mysql_store_result(conn) ;

        while((class_row = mysql_fetch_row(class_results_set)) != NULL)
                {
                // We have a class for which CBO will now be calculated

                SQL_build_methods_table(query, class_row[0]) ;

                handle_mysql_query(conn, query) ;

                SQL_build_methods_data_table(query, class_row[0]) ;

                handle_mysql_query(conn, query) ;

                SQL_build_all_data_table(query, class_row[0]) ;

                handle_mysql_query(conn, query) ;

                SQL_build_class_links_table(query, class_row[0]) ;

                handle_mysql_query(conn, query);

                SQL_select_class_links_table(query, class_row[0]) ;

                handle_mysql_query(conn, query);

                printf("Class %s has links to the following classes: \n\n",
                        class_row[1]) ;

                // For the class in question e.g. class ID = 1, we now have a list of
                // all class IDs linked to class 1, for which CBO will now be
                // calculated.

                linked_class_results_set = mysql_store_result(conn) ;

                cbo = 0 ;

                while((linked_class_row = mysql_fetch_row(linked_class_results_set))!=
                                                                                NULL)
                        {
                        printf("To class: %s \n", row[1]) ;
                        cbo++ ;
                        } ;

                printf("\n CBO metric for class %s is %i \n\n", class_row[1], cbo) ;

                // Finished calculating CBO for this class so release the results set
                mysql_free_result(linked_class_results_set) ;
```

```
                }  // End of while fetch class name loop
        // Finished claculating CBO metric for all classes so release the results set

        mysql_free_result(class_results_set) ;

        } ;  // End of function calculate_CBO

// ---------------------------------------------------------------------------




void SQL_select_class_list (char * qry)
        {
        // SELECT * FROM Class

        strcpy(qry, "SELECT * FROM Class") ;
        } ;

// ---------------------------------------------------------------------------

void SQL_build_methods_table(char *qry, char *cls_id)
        {
        // For class ID = 1, this query creates a table called "methods_in_class1"
        // which contains a list of all methods in class 1. The table has just one
        // field which is called: method_id.

        strcpy(qry, "CREATE TEMPORARY TABLE methods_in_class") ;
        strcat(qry, cls_id) ;
        strcat(qry, " SELECT is_class_method_of.method_id FROM is_method_of WHERE
                is_class_method_of.class_id = ") ;
        strcat(qry, cls_id) ;
        } ;

// ---------------------------------------------------------------------------

void SQL_build_methods_data_table(char *qry, char *cls_id)
        {
        // For class ID = 1, this query creates a table called "all_data_for_class1"
        // which contains a list of all data attributes contained in member function
        // parameter lists or as local data declared within the member functions
        // belonging to class1. The resultant table has the following structure:
        // data_id.

        strcpy(qry, "CREATE TEMPORARY TABLE all_data_for_class") ;
        strcat(qry, cls_id) ;
        strcat(qry, " SELECT is_local_method_data_of.data_id FROM is_local_method_data_of,
                is_class_method_of") ;
        strcat(qry, cls_id) ;
        strcat(qry, " WHERE is_local_method_data_of.method_id = is_class_method_of") ;
        strcat(qry, cls_id) ;
        strcat(qry, ".method_id") ;
        } ;

// ---------------------------------------------------------------------------

void SQL_build_all_data_table(char *qry, char *cls_id)
        {
        // For class ID = 1, this query augments the table called "all_data_for_class1"
        // with additional data extracted from the class itself, i.e. instance and
        // class level data members.

        strcpy(qry, "INSERT INTO all_data_for_class") ;
        strcat(qry, cls_id) ;
        strcat(qry, "(data_id) SELECT is_class_data_of.data_id FROM is_class_data_of WHERE
                is_class_data_of.class_id = ") ;
        strcat(qry, cls_id) ;
        } ;

// ---------------------------------------------------------------------------

void SQL_build_class_links_table(char *qry, char *cls_id)
        {
        // For class ID = 1, this query creates a table called "links_class1"
        // containing a list of classes to which class1 is coupled. Table links_class1
        // has the following structure: class_id, class_name.

        strcpy(qry, "CREATE TEMPORARY TABLE links_class") ;
        strcat(qry, cls_id) ;
        strcat(qry, " SELECT DISTINCT is_instance_of.class_id, Class.class_name FROM
                is_instance_of, Class, all_data") ;
        strcat(qry, cls_id) ;
        strcat(qry, " WHERE all_data_for_class") ;
        strcat(qry, cls_id) ;
        strcat(qry, ".data_id = is_instance_of.data_id AND is_instance_of.class_id =
                Class.class_id") ;
        } ;

// ---------------------------------------------------------------------------

void SQL_select_class_links_table(char *qry, char *cls_id)
        {
        strcpy(qry, "select * from links_class");
        strcat(qry, cls_id);
        } ;

// ---------------------------------------------------------------------------
```