

# A Software Metric System for Module Coupling

A. Jefferson Offutt  
Mary Jean Harrold  
Priyadarshan Kolte

Department of Computer Science  
Clemson University  
Clemson, SC 29634-1906  
phone: 803-656-5882  
email: ofut@cs.clemson.edu

## Abstract

Low module coupling is considered to be a desirable quality for modular programs to have. Previously, coupling has been defined subjectively, and not quantified, making it difficult to use in practice. In this paper, we extend previous work to reflect newer programming languages, and quantify coupling by developing a general software metric system that allows us to automatically measure coupling. We have precisely defined the levels of coupling so that they can be determined algorithmically, incorporated the notion of direction into the coupling levels, and accounted for different types of non-local variables present in modern programming languages. With our system, we can measure the coupling between all pairs of modules in a system, measure the coupling of a particular module with all other modules in a system, and measure the coupling of an entire system. We have implemented our metric system so that it measures the coupling between pairs of procedures in arbitrary C programs and have analyzed several well-used systems of various sizes.

*Keywords:* software design, coupling, data flow, program slicing.

# 1 Introduction

Software complexity measures are meant to indicate whether the software has desirable attributes such as understandability, testability, maintainability, and reliability. As such, they may be used to suggest parts of the program that are prone to errors. An important way to reduce complexity is to increase modularization [10]. As part of their structured design methodology, Constantine and Yourdon [2] suggested that modularity of software design be measured with two qualitative properties: cohesion and coupling. *Cohesion* describes a module's functionality; the highest degree of cohesion is obtained when a module performs one function. On the other hand, *coupling* is the degree of interdependence between pairs of modules; the minimum degree of coupling is obtained by making modules as independent as possible. Ideally, a well designed software system maximizes cohesion and minimizes coupling<sup>1</sup>. Page-Jones [13] gives three principle reasons why low coupling between modules is desirable: (1) fewer interconnections between modules reduce the chance that a fault in one module will cause a failure in other modules, (2) fewer interconnections between modules reduce the chance that changes in one module cause problems in other modules, which enhances reusability, and (3) fewer interconnections between modules reduce programmer time in understanding the details of other modules.

Myers [10] refined the concept of coupling by presenting well-defined, though informal, levels of coupling. Since his levels were neither precise nor prescriptive definitions, coupling could only be determined by hand, leaving room for subjective interpretations of the levels. Other researchers [15, 8, 6, 14] have used coupling levels or similar measures to evaluate the complexity of software design and relate this complexity to the number of software faults. Recently, Fenton and Melton [4] developed a measurement theory that provides a basis for defining software complexity and used hand-derived coupling measures to demonstrate their theory. They enhanced previous work in coupling by incorporating the number of interconnections between modules into the measure and by considering the effects on coupling of return values and reference parameters as well as input parameters.

However, module coupling is still considered to be an imprecise measure of software complexity. Jalote states, "Coupling is an abstract concept and is as yet not quantifiable" [7]. In this paper, we extend previous work to reflect newer programming languages, and quantify coupling by developing a general software metric system that allows us to automatically measure coupling. We offer precise definitions of the coupling levels so that they can be determined algorithmically, incorporate the notion of direction into the coupling levels, and account for different types of non-local variables present in modern programming languages. With our system, we can measure the coupling between all pairs of modules in a system, measure the coupling of a particular module with all other modules in a system, and measure the coupling of an entire system.

Our metric system first determines the variable use information needed to compute the coupling measure using a mixture of textual inspection, data flow analysis, and a restricted form of program slicing [16]. Next, the system identifies the levels of coupling and numbers of interconnections among the modules. The central component of the system uses an algorithm that precisely identifies the level of coupling and the number of interconnections of that level between two modules. Finally, the system uses the coupling levels and numbers of interconnections to compute a coupling measure that is based on a complexity metric that is similar to Fenton and Melton's [4]. We have implemented our metric system to identify coupling measures for modules

---

<sup>1</sup> We follow earlier researchers and consider a *module* to be a single procedure/function, rather than a collection of procedures.

written in the C programming language and have analyzed several well-used systems of various sizes.

Our software metric system fulfills Kafura and Henry's [8] four criteria for a practical and powerful software metric system. First, our system is usable in a large-scale system environment, as demonstrated by our measurements of several large-scale systems such as the Free Software Foundation, Inc's C compiler (GCC)<sup>2</sup>. Second, our metrics are complete and sensitive to changes in the structure of the system since any change in the interconnections among modules changes the coupling measure. Third, our software metric system has been applied to actual large-scale systems. Finally, our measurement is robust since it applies to any system structure and our results are easily interpretable.

In the next section, we present Myer's full list of coupling levels and discuss some of the recent work on coupling. In section 3, we present our precise model of coupling levels, our generalized metric system, and our implementation. Section 4 presents some experimental results from this implementation and our interpretation of these results. In section 5.1, we discuss several ideas for future work, including ideas for extending our metric system to handle modules containing more than one procedure. Finally, concluding remarks are given in section 6.

## 2 Coupling

Myers [10] defined six distinct levels of coupling to measure the interdependence among the modules; we include no coupling as the zeroth level. The coupling levels were ordered by Page-Jones [13] according to their effects on the understandability, maintainability, modifiability and reusability of the coupled modules. If two modules are coupled in more than one way, they are considered to be coupled at the highest level:

0. **Independent Coupling** – No coupling between the modules.
1. **Data Coupling** – Two modules are **data** coupled if they pass data through scalar or array parameters.
2. **Stamp Coupling** – Two modules are **stamp** coupled if they pass data through a parameter that is a record. **Stamp** coupling is perceived as worse than **data** coupling because any change to the record will affect *all* of the modules that refer to that record, *even those modules that do not refer to the fields that are changed*.
3. **Control Coupling** – Two modules are **control** coupled if one passes a *flag* value that is used to control the internal logic of the other.
4. **External Coupling** – Two modules are **external** coupled if they communicate through an external medium such as a file.
5. **Common Coupling** – Two modules are **common** coupled if they refer to the same global data.
6. **Content Coupling** – Two modules are **content** coupled if they access and change each other's internal data state or procedural state.

Page-Jones [13] also introduced the notion of **tramp coupling**, where data may flow through many intermediate modules from where the data are defined to where they are used. This differs from the other coupling levels in that it measures the coupling among many modules instead of just two modules. **Tramp** coupling is usually a pathological form of **data** coupling, but can also be **stamp** or **control** coupling.

Since Myer's initial work on coupling, several researchers have used coupling or related measures to evaluate software. Troy and Zweben [15] used coupling to measure the quality of a design by relating the

---

<sup>2</sup>©1987, 1989 Free Software Foundation, Inc., 675 Mass Avenue, Cambridge, MA 02139.

level of coupling to the number of faults in the software. Their experimental study derived coupling measures from a set of designs, kept track of the modifications to the software, and performed a data analysis of the measures and the changes. Their principal result was that module coupling was a very important factor in quality, and a good predictor of the number of faults in software. In contrast to their study where coupling levels were determined by hand, our technique is easily automated. Thus, our algorithms not only make experimentation of this type easier, but can be useful to predict which modules will contain the most faults. On the other hand, they evaluated design documents, allowing them to compute coupling levels before implementation, whereas our implementation requires the code to be written (the exact information we need is detailed in section 3.2).

Kafura and Henry [8] suggested a similar method for measuring software based on *information flow*. They derived their metrics from procedures by considering the flow of information to input parameters, output parameters, and global data structures. They defined *global flows* to be the data flowing through global data structures, *direct local flows* to be data flowing through procedure parameters, and *indirect local flows* to be data that is passed from one procedure to another *through* another procedure (i.e., A to B to C). Their information flow types are very similar to coupling levels, but some of their flow types include several coupling levels. Specifically, *indirect local flows* correspond to **tramp coupling**, *global flows* include **common** coupling, and *direct local flows* include **data**, **stamp** and **control**. Because the coupling levels that we measure are at a finer level of detail, we are extending Kafura and Henry's work in a way that should result in a more precise measure of software quality.

Hutchens and Basili [6] proposed the use of *clustering* with *data bindings* for measuring software quality. A *potential data binding* occurs when a variable is in the scope of two procedures (e.g.,  $x$  is in the scope of P and Q). A *used data binding* occurs when the two procedures use the variable. An *actual data binding* is a used data binding where one procedure defines the variable and the other procedure uses it. A *control flow data binding* is an actual data binding where control passes from the defining procedure to the using procedure. Hutchens and Basili used these ideas to measure the complexity of software. Like Kafura and Henry, they derive their measures from program source, but do not derive as detailed a measurement as does coupling. Specifically, data bindings do not differentiate between data computation and control uses, and do not differentiate parameter values from global structures.

Selby and Basili [14] used Hutchens and Basili's data binding measurements [6] to measure a 148,000 line system from a production environment. They collected error data during the testing phase, and applied five tools to calculate data bindings using clustering. They found that procedures with the highest coupling had up to seven times as many faults as procedures that exhibited low coupling. Again, our algorithms compute a more precise coupling measure, extending Hutchens, Selby, and Basili's work to get a more precise measure of software quality.

Fenton and Melton [4] presented a way to measure the structure of software and applied it using coupling. They described a measure of complexity of software design that used hand-derived pairwise coupling between modules. They used only the highest level of coupling between two modules, and included the number of interconnections of that level between the two modules. If  $i$  is the measure of the greatest coupling type, and  $n$  is the number of interconnections between two modules  $x$  and  $y$  of type  $i$ , then they defined the following

as an ordinal measure of coupling between  $x$  and  $y$ :

$$\mathbf{Equation\ 1} : M(x, y) = i + \frac{n}{n + 1}.$$

Whereas this gives a pairwise coupling between modules, Fenton and Melton suggested using the median of all pairwise module couplings to measure global coupling of a system. While their work gives a way to use coupling levels to measure the quality of software, we present algorithms for determining coupling levels. We expect our results to be used in measures such as theirs by using our algorithms to provide the pairwise coupling measures to supply numbers to be used in their complexity measure.

### 3 A Technique to Measure Coupling

In this section, we present a coupling model that extends previously defined coupling levels in several ways. The original coupling levels are insufficient because of recent advances in programming languages and design practices and because there were several cases of potential coupling that were not included in the original levels. First, we define several types of `global` coupling to reflect scoping features in modern programming languages (such as *non-local* variables). Second, we incorporate the notion of *bidirection* into our coupling model to measure both in and out couplings between modules. Although Fenton and Melton [4] include bidirectional coupling, the earlier researchers, Myers [10], Page-Jones [13], and Constantine and Yourdon [2] made no mention of coupling via return values. Third, we extend coupling levels to include a combination of data and control coupling, introducing levels that were merged with data coupling in Myer's levels. Fourth, we identify coupling between modules that call each other without passing data or sharing variables, a case which was missing in previous definitions of coupling. Fifth, we give prescriptive definitions of the coupling levels so that they can be determined algorithmically. We next discuss our software metric system that measures coupling according to this model, and then describe our implementation of the system.

#### 3.1 Extensions to Coupling Levels

Most of the literature on software coupling makes no distinction between data passed *in* to a module and data passed *out* of a module through reference parameters and return values, even though the two directions may display different levels of coupling. For example, consider the following module pair:

```
module P                                module Remainder (N, div)
  if (Remainder (X,2) > 0) then          quo = N/div
    Z = 2*X                              rem = N - (div*quo)
  else                                    return (rem)
    Z = 2/X                                end Remainder
  endif
end P
```

Since `X` in module `P` is passed to module `Remainder` as a data value, modules `P` and `Remainder` are **data** coupled. However, the value returned from `Remainder` to `P` is used to control the logic of `P`, resulting in **control** coupling between the two modules. We distinguish between inputs and outputs, and consider *bidirectional coupling* since we measure coupling into, as well as out of, a module. In the above example, the in-coupling between `P` and `Remainder` is **data**, but the out-coupling is **control**. For some types of coupling,

the in-coupling and the out-coupling are always the same. For example, two modules that share global variables have the same in-coupling and out-coupling. We call coupling levels that are both in-coupled and out-coupled *commutative*.

Another issue that has not been previously addressed is the case where a parameter is used as a data value to define a variable that later is used to define a flag. For example, consider the following piece of code:

```

module P (X)
  Z = X * X - X;
  if (Z > 0)
    return (X);
  else
    return (1);
  endif
end P

```

In this case, **X** appears to be used as a data value in module **P**, which means that any module that calls module **P** would be **data** coupled with it. However, **X** is indirectly used as a **control** value through assignment to **Z**. We classify this hybrid of **data** and **control** coupling as **data/control** coupling.

Although previous definitions of coupling levels have distinguished between scalar parameters that are used as data values and scalar parameters that are used as control values, no similar distinction has been made in the use of record parameters (**stamp**). Since records can be used as either data values or control values, we define the following hybrid levels of coupling: **stamp-data** coupling, **stamp-control** coupling, and **stamp-data/control** coupling.

With the scoping rules that exist in modern programming languages, all global variables are not visible to all modules in the system. If a variable **X** is global to a restricted set of modules **A**, **B**, and **C**, but not to all the modules in the system, then we say that **X** is a *non-local* variable, and **A**, **B**, and **C** are **non-local** coupled. This situation often arises with data abstraction. For example, in a **stack** data type, a non-local such as a stack pointer is accessed only by the stack operator functions. If **X** is global to all modules in the system, then it is *global*, and all modules that use **X** are **global** coupled.

Previously defined coupling levels considered two modules to be independently coupled if no data were passed or shared between them. However, we found it necessary to distinguish between modules that have no relationship at all, and modules that call each other but do not share any parameters or other data. We call this new level **call** coupling, since the modules are coupled only through a call.

To precisely define our coupling levels, we classify each call and return parameter by the way it is used in the module. We borrow the classification of uses as computation-uses (C-uses) and predicate-uses (P-uses) from data flow testing [5] and define indirect-uses (I-uses). A C-use occurs whenever a variable (or parameter) is used in an assignment or output statement. A P-use occurs whenever a variable is used in a predicate statement. An I-use occurs whenever a variable is a C-use that affects some predicate in the module. We define our precise coupling levels between two modules **A** and **B** in the following list and indicate which of the coupling levels are bidirectional and which are commutative.

0. **Independent Coupling** (commutative) - **A** does not call **B** and **B** does not call **A**, and there are no common variable references or common references to external media between **A** and **B**.
1. **Call Coupling** (commutative) - **A** calls **B** or **B** calls **A** but there are no parameters, common variable references or common references to external media between **A** and **B**.

2. **Scalar Data Coupling** (bidirectional) - A scalar variable in **A** is passed as an actual parameter to **B** and it has a C-use but no P-use or I-use.
3. **Stamp Data Coupling** (bidirectional) - A record in **A** is passed as an actual parameter to **B** and it has a C-use but no P-use or I-use.
4. **Scalar Control Coupling** (bidirectional) - A scalar variable in **A** is passed as an actual parameter to **B** and it has a P-use.
5. **Stamp Control Coupling** (bidirectional) - A record in **A** is passed as an actual parameter to **B** and it has a P-use.
6. **Scalar Data/Control Coupling** (bidirectional) - A scalar variable in **A** is passed as an actual parameter to **B** and it has an I-use but no P-use.
7. **Stamp Data/Control Coupling** (bidirectional) - A record in **A** is passed as an actual parameter to **B** and it has an I-use but no P-use.
8. **External Coupling** (commutative) - **A** and **B** communicate through an external medium such as a file.
9. **Non-Local Coupling** (commutative) - **A** and **B** share references to the same non-local variable; a non-local variable is visible to a subset of the modules in the system.
10. **Global Coupling** (commutative) - **A** and **B** share reference to the same global variable; a global variable is visible to the entire system.
11. **Tramp Coupling** (bidirectional) - A formal parameter in **A** is passed to **B** as an actual parameter, **B** subsequently passes the corresponding formal parameter to another procedure without **B** having accessed or changed the variable.

We used Page-Jone's original ordering [13] to develop our coupling levels. We inserted **Call** coupling before all levels that involve data passing. We added **scalar** and **stamp data/control** coupling after the control couplings because they exhibit the worst of both **data** and **control** coupling. **Data/control** couplings are also difficult for programmers to recognize because of the indirect nature of the control aspect, which increases the effort needed to understand, maintain, and modify the program. **Non-local** coupling is a restricted case of **global** coupling so it is placed just before **global** coupling.

Finally, **tramp** coupling is placed after **global** because, like **global** coupling, it is a way to share data between modules that are not directly connected through a procedure call. Making a change to a data item involved in **tramp** coupling requires making changes to an arbitrary number of modules, many of which do not actually modify or use the data item, but only pass the data item to another module, whereas the same changes to a data item involved in **global** coupling requires changes only to modules that actually modify or use the data item. For example, **tramp** coupling often arises from run-time parameters that are processed by a main program module, but used by a low-level module. This results in tramp coupling when a variable is set by the main module and passed through intervening modules to the low-level module that uses the variable, and **global** coupling when a global variable is set by the main module and used by the low-level module.

## 3.2 Our Metric System

Our software metric system incorporates the coupling model we defined above to determine coupling measures for each pair of modules. Our algorithms are based on the generation and use of data information.

To determine the coupling level in which a pair of modules fall, we consider 5 orthogonal categories of information:

1. How is the data used? (E.g., predicate use, computation.)
2. How much of the available data is used? (E.g., the entire record, or just one field.)
3. What is the scope of the data?
4. How far is the data propagated?
5. Is the direction of data flow into or out of the called module?

To determine this information, we require at least a detailed design, and the form must be such that the design is parsable (i.e., flowcharts or pseudocode).

Our system uses three major steps. For each module being considered, we first use the algorithm **GetSummary** to produce a summary file that contains the information about the module needed to measure its coupling with other modules. Next, our algorithms use the summary files to compute the levels of coupling, both **in** and **out**, and the number of interconnections at that level for each pair of modules being considered. Here, we present only algorithm **FindInCoupling** and omit the analogous algorithm **FindOutCoupling**. Finally, the coupling levels and interconnection numbers are used to get an overall coupling measure for the system. Although any coupling measure can be used, we present and use an adjusted version of that given by Fenton and Melton [4].

Though we discuss a software metric system to compute the coupling between pairs of modules, the resulting measures are easily combined to gain a coupling measure for an integrated system. To find the coupling for a program or software system, the first step of computing the summary files for all modules and the second step of finding the coupling between pairs of modules is required as before. Only the third step of determining the coupling measure is modified. The median (or some other appropriate statistic) of pairwise coupling measures can be used to measure either the coupling between a particular module and all others or the coupling for the entire program/system.

### 3.2.1 Step One: Getting the Summary Files

Algorithm **GetSummary**, given in Figure 1, computes the summary information for a module by using a combination of textual inspection, data flow analysis, and a restricted form of forward program slicing. First, **GetSummary** makes one pass over the text of **A** and identifies the definitions and uses of all variables, the call sites, the parameters and their types, and all predicate statements. Next, **GetSummary** performs data flow analysis using the definition and use information to obtain the sets of reachable uses of all definitions in **A**. Since each module is an individual procedure, we use traditional intraprocedural data flow analysis techniques [1] to compute the reachable use sets. For each definition, a **du\_set** is constructed that contains all reachable uses of that definition.

Finally, **GetSummary** uses the data flow information and a forward slicing technique to classify each formal parameter as having a C-use, a P-use or an I-use. For each formal parameter **f**, **GetSummary** considers the **du\_set** of the implicit definition of **f** at the entry to **A** and uses it to classify the formal parameter. If the

```

algorithm  GetSummary (A);
input     code for module A
output    a summary file for module A
declare   pred_stats      : set of predicate statements
          visited_defs    : set of statements that have been visited
          trans_defs      : set of statements that use the defs in visited_defs
          du_set          : set of uses of a definition
          change, found   : boolean
          cur_def         : statement number
          f               : formal parameter

begin
  /* Scan A's code */
  Get definitions/uses of all variables, scalar/record parameters and predicate statements

  /* Perform dataflow analysis */
  Get a du_set for each definition in A

  /* Perform Forward Slicing */
  pred_stats := {predicate statements}
  foreach formal param f do
    cur_def := definition of f at entry to A
    if (du_set[cur_def] =  $\Phi$ ) then
      f has no uses in A
    elseif (du_set[cur_def]  $\cap$  pred_stats)  $\neq$   $\Phi$  then
      f has a P-use
    else
      visited_defs :=  $\Phi$ 
      trans_defs := {cur_def}
      change := true; found := false
      while (change  $\wedge$  not(found)) do
        change := false
        trans_defs := trans_defs  $\cup$  du_set[cur_def]
        if (trans_defs  $\cap$  pred_stats)  $\neq$   $\Phi$  then
          found := true
        else
          visited_defs := visited_defs  $\cup$  {cur_def}
          if (trans_defs - visited_defs)  $\neq$   $\Phi$  then
            cur_def := an unvisited def in trans_defs
            change := true
          endif
        endif
      endwhile
      if (found) then
        f has an I-use
      else
        f has a C-use
      endif
    endif
  endfor
end GetSummary

```

Figure 1. Algorithm GetSummary uses textual inspection, data flow analysis and a restricted forward slicing technique to get a summary file for module A.

`du_set` is empty, `f` is marked as having no uses. Otherwise, the `du_set` is examined for P-uses and marked accordingly if any are found.

At this point, an unmarked formal parameter has either a C-use or an I-use. To determine whether the formal parameter has an I-use, `GetSummary` attempts to find a P-use of `f` in the transitive closure of the definition and use relationships of the implicit definition of `f` at the entry to `A`. A set `trans_defs` contains those definitions to be processed and initially contains this implicit definition of `f`. For each definition `cur_def` in `trans_def`, its `du_set` is examined for a P-use. If one is found, the processing stops and `f` is marked as an I-use. Otherwise, processing continues by adding all statements (definitions) corresponding to the uses in `du_set` to `trans_defs`. A set `visited_defs` keeps track of definitions previously processed so that each definition is processed only once. If all definitions in the transitive closure are processed without finding a P-use, `f` is marked as a C-use.

### 3.2.2 Step Two: Finding the Coupling Levels/Interconnection Numbers

After computing the summary files for modules `A` and `B`, our metric system uses algorithm `FindInCoupling`, given in Figure 2, to determine the in-coupling level and number of interconnections of that level between the modules. `FindInCoupling` looks for the level of coupling between modules `A` and `B` from the highest level (`tramp`) to the lowest level (`independent`) and counts the number of interconnections of the highest level found. Since we define the coupling level as the highest level between modules `A` and `B`, the algorithm stops when it identifies a coupling level.

To identify `tramp` coupling, `FindInCoupling` searches for cases where a formal parameter of `A` is passed as a parameter to `B` and the only uses of the corresponding formal parameter in `B` are to pass it to some other procedure that `B` calls. The algorithm counts the number of parameters that are passed as `tramp`. After eliminating the case of `tramp` coupling between `A` and `B`, the algorithm checks for `global` coupling by considering the intersection of the `Globals` sets for `A` and `B` in the summary files. If a nonempty intersection is found, the level is set and the number of interconnections is the cardinality of the intersection of the `Globals` sets. If there is no `global` coupling, a similar technique is used to identify `non-local` or `external` coupling.

If none of the above coupling levels is found, `FindInCoupling` considers the callsites to `B` in `A`. For each formal parameter in `B`, it looks for the highest level of coupling among `stamp data/control`, `scalar data/control`, `stamp control`, `scalar control`, `stamp data`, and `scalar data`; it also counts the number of interconnections at that level. A procedure `Set` is used to record the level and number. If no callsites to `B` in `A` pass parameters, the two modules are `call` coupled. Finally, if none of the coupling levels is found, `A` and `B` are `independently` coupled.

### 3.2.3 Step Three: Computing the Coupling Measure

Step Three uses the results of Step Two to compute the measure `M` of coupling between `A` and `B`. Any measure that uses the coupling level and number of interconnections can be used. We adjust Fenton and Melton's measure [4] that incorporates the coupling level with the number of interconnections, which was given in section 2. For a coupling level,  $i$ , the coupling measure is proportional to the number of interconnections  $n$ . However, with their definition, the measure is always in the interval  $[i + \frac{1}{2}, i + 1)$ , which places the value for coupling level  $i$  close to  $i + 1$  than  $i$ . Intuition suggests that the effect of the number of interconnections

```

algorithm FindInCoupling (A, B);
input      A, B: modules with  $a_1, \dots, a_m$  and  $b_1, \dots, b_n$  formal parameters respectively
output     level := 0 : [0..11]; number := 0: integer

begin
  foreach B( $a_i$ )  $\in$  CallSites(A) do
    foreach C( $b_i$ )  $\in$  CallSites(B), where  $a_i$  in A is bound to  $b_i$  in B
      if  $b_i$  has no uses in B then
        level := 11                                     /* tramp */
        number := number + 1
      endif
    endfor
  endfor
  if level = 0 then                                     /* not tramp */
    if (Globals (A)  $\cap$  Globals (B))  $\neq$   $\Phi$  then      /* global */
      level := 10
      number := | Globals (A)  $\cap$  Globals (B) |
    elseif (NonLocals(A)  $\cap$  NonLocals(B))  $\neq$   $\Phi$  then /* non-local */
      level := 9
      number := | NonLocals (A)  $\cap$  NonLocals (B) |
    elseif (Externals (A)  $\cap$  Externals (B))  $\neq$   $\Phi$  then /* external */
      level := 8
      number := | Externals (A)  $\cap$  Externals (B) |
    elseif B  $\in$  CallSites (A) then
      if A passes parameters to B then
        foreach  $b_i \in b_1, \dots, b_n$  do
          if (IsRecord( $b_i$ )  $\wedge$  I-use( $b_i$ )) then Set (level, number, 7) /* stamp data/control */
          elseif (IsScalar( $b_i$ )  $\wedge$  I-use( $b_i$ )) then Set(level, number, 6) /*scalar data/control*/
          elseif (IsRecord( $b_i$ )  $\wedge$  P-use( $b_i$ )) then Set(level, number, 5) /* stamp control */
          elseif (IsScalar( $b_i$ )  $\wedge$  P-use( $b_i$ )) then Set(level, number, 4) /* scalar control */
          elseif (IsRecord( $b_i$ )  $\wedge$  C-use( $b_i$ )) then Set(level, number, 3) /* stamp data */
          elseif (IsScalar( $b_i$ )  $\wedge$  C-use( $b_i$ )) then Set(level, number, 2) /* scalar data */
          endif
        endfor
      else
        level := 1                                     /* call */
      endif
    endif
  endif
  return(level, number)                               /* if level is still 0, independent */
end FindInCoupling

procedure Set(level, number, newlevel);
input      level, number, newlevel
output     level, number

begin
  if (level = newlevel) then
    number := number + 1
  elseif (level < newlevel) then
    level := newlevel
    number := 1
  endif
end Set

```

end FindInCoupling

```

procedure Set(level, number, newlevel);
input      level, number, newlevel
output     level, number

begin
  if (level = newlevel) then
    number := number + 1
  elseif (level < newlevel) then
    level := newlevel
    number := 1
  endif
end Set

```

Figure 2. Algorithm FindInCoupling finds the in-coupling level and number of interconnections of that level for modules A and B.

$n$  on the coupling level  $i$  should fall closer to  $i$  than  $i + 1$ , so we refine Fenton and Melton’s measure by subtracting  $\frac{1}{2}$ :

$$\mathbf{Equation\ 2}: M(x, y) = i + \frac{n}{n + 1} - \frac{1}{2} = i + \frac{n - 1}{2(n + 1)}.$$

This measure guarantees that  $M$  will be in the interval  $[i, i + \frac{1}{2})$ . This change has no effect on the relative orderings of any values, but is more intuitively satisfying, and makes it easier to interpret the values.

First, the in-coupling measure  $\mathbf{M}_1$  is computed by considering the level  $i$  and number of connections  $n$  for the call of **A** to **B** and the out-coupling measure  $\mathbf{M}_2$  is computed by considering the level  $i$  and the number of connections  $n$  for the return of **B** to **A**. Both  $\mathbf{M}_1$  and  $\mathbf{M}_2$  are computed with our newly defined measure  $i + \frac{n-1}{2(n+1)}$  that ensures that we get a measure between  $i$  and  $i + \frac{1}{2}$ . Finally, the system computes  $\mathbf{M}$ , the coupling measure between modules **A** and **B**, as the maximum of the in-coupling measure  $\mathbf{M}_1$  and the out-coupling measure  $\mathbf{M}_2$ .

### 3.3 Implementation

We implemented our software metric system in a system that identifies the coupling between pairs of procedures for C programs. Our implementation is incorporated into the GCC compiler and consists of three programs containing about 3000 lines of C code. The user specifies the system to be analyzed and the first program implements algorithm **GetSummary** to produce the summary files for all procedures in the system. The second program permits the user to choose a set of modules in the system for analysis. The main function of this program is to build tables to permit efficient cross references to global variables and procedure names.

Finally, the third program implements algorithm **FindInCoupling** for all pairs of modules chosen. Our implementation does not measure **external** coupling and considers *static* global variables in C programs to be non-locals. For each pair of procedures, the **level** and the **number** are computed by **FindInCoupling** and then used to find the in-coupling measure  $\mathbf{M}_1$ . We have implemented only the in-coupling measure  $\mathbf{M}_1$ ; the out-coupling measure  $\mathbf{M}_2$  can be implemented similarly. Thus, in our experiments,  $\mathbf{M}_1$  represents the coupling measure between the two procedures.

## 4 Experimentation

We chose a variety of C programs and systems (collections of programs) for our experiments, with sizes ranging from 7000 lines to 90,000 lines of source. SC is a relatively small public domain spreadsheet program that was originally written by James Gosling. Make is the GNU version 3.60 release of the UNIX dependency-compilation utility. GCC is release 1.37.1 of the GNU C compiler, a large and recently written production quality program. Mothra [3, 9] is a mutation-based software testing system that is composed of about a dozen separate programs, or *tools*, all of which use a common library. It tests programs written in Fortran 77. IMSCU [12, 11] is a smaller-scale mutation testing system designed for educational purposes that tests programs written in a subset-Pascal language. Like Mothra, IMSCU consists of several tools that use a common library, but has fewer capabilities, is much smaller, corrects some perceived design deficiencies of Mothra, and was designed with greater care towards modularity than Mothra. We included both to be able to compare two systems with intersecting functionalities (Mothra includes all functions of IMSCU, but

has more tools) and included only the tools that the two systems have in common. Characteristics of our experimental programs are shown in Table 1.

PROGRAM	FILES	PROCS	LINES
SC	11	193	7613
Make	21	157	12783
GCC	47	1134	90936
IMSCU	16	233	8934
Mothra	38	533	36692

**Table 1.** Experimental Programs

In Table 1, the FILES column gives the number of files in the programs, PROCS gives the number of procedures, and LINES gives the number of source lines (a rough estimate that includes comments and blank lines). We present coupling data from these programs in several formats. For each of our programs, Table 2 shows how many module pairs were coupled for each of our coupling levels. Our implementation does not include **external** coupling, so this data is not shown. Although our implementation measures **call** coupling, the data has been combined with **independent** coupling so as to fit in the table. We evaluated SC and Make as single programs and separated GCC, IMSCU, and Mothra into several subsystems. The six subsystems and the library for Mothra and IMSCU are roughly similar in functionalities, and we excluded several tools of Mothra that are not present in IMSCU.

Pgm	Sub System	Indep/ Call	Scalar Data	Stamp Data	Scalar Control	Stamp Control	Scalar D/C	Stamp D/C	Non-local	Global	Tramp
SC		14433	44	7	44	12	47	2	38	3882	19
Make		11696	139	3	103	10	67	36	35	132	25
GCC	cccp	2120	44	0	5	2	45	14	1	466	4
	parser	28742	129	48	63	42	11	180	295	1834	31
	rtlgen	104359	153	62	30	31	10	326	230	10184	55
	backend	130526	45	21	31	75	11	262	547	13433	40
IMSCU	Library	13896	75	9	36	0	11	5	823	544	1
	Equiv	14068	76	9	38	0	11	6	823	544	1
	Decomp	15808	106	14	50	0	16	7	833	555	2
	MutGen	16560	95	11	45	0	12	8	844	568	2
	TCEnt	14933	92	17	43	0	11	6	823	545	1
	Interp	16938	108	23	43	0	12	7	839	557	1
	Stats	15108	79	16	39	0	14	7	823	566	1
Mothra	Library	51747	105	49	110	6	83	7	731	1432	15
	Equiv	52075	105	49	112	6	83	7	731	1432	15
	Decomp	59621	128	51	130	6	104	12	783	1625	21
	MutGen	59316	136	62	135	7	99	10	731	1970	15
	TCEnt	54036	133	49	113	6	87	8	731	1438	15
	Interp	62433	157	78	145	6	120	10	732	1641	19
		Stats	59007	118	54	115	6	101	9	731	1617

**Table 2.** Numbers of Module Pairs for Each Coupling Level.

Table 2 shows that the vast majority of all module pairs in each system are **independent** coupled or **call** coupled. It is natural to expect most of the modules in systems of this size to be unrelated. The next largest coupling level for SC, GCC, and Mothra is **global** coupling. Both GCC and IMSCU had more **non-local** coupled module pairs than module pairs coupled at any other level, which is probably an indication of heavy use of abstract data types (ADT), where the data structure is global to procedures in the ADT but not the rest of the system. Use of data abstraction was a design goal for IMSCU, as well as for Mothra, which has

**non-local** coupling as its third most prevalent coupling level. Make was the only program with a larger number of module pairs coupled at the **scalar data** level than at an other level, although almost as many module pairs in it were **global coupled**.

Since extensive use of global data is widely considered to be dangerous [17], these results are disturbing. They seem to indicate that our theoretical ideas of good software engineering habits are not followed in practice as well as we would like, and perhaps that they are more difficult to follow than we would expect.

We included both Mothra and IMSCU so that we could compare similar systems, but since Mothra is so much bigger, they are difficult to compare based on the numbers shown in Table 2. Table 3 shows the percentages of each level of coupling for both Mothra and IMSCU. For each common subsystem, the percentages for IMSCU are adjacent to those for Mothra. The percentages of uncoupled module pairs are consistently between 90 and 92% for the IMSCU subsystems, while they are between 95 and 96% for the Mothra subsystems. This is probably because Mothra is bigger and contains more files (38 as opposed to 16, as shown in Table 1). The IMSCU subsystems (except for MutGen) have a slightly higher percentage of **global coupled** module pairs, more **scalar data coupling**, but fewer **tramp coupling**. In Mothra, many of the command line parameters are used to set flags that are passed down through the call hierarchy to the procedures where they are used; IMSCU replaces most of these **tramps** with global flags. The IMSCU subsystems also have significantly more **non-local** coupled pairs, indicating their higher reliance on abstract data types. Thus, we feel that the design goal of making IMSCU more modular than Mothra was met.

Pgm	Sub System	Indep/ Call	Scalar Data	Stamp Data	Scalar Control	Stamp Control	Scalar D/C	Stamp D/C	Non-local	Global	Tramp
Library	IMSCU	90.23	0.49	0.06	0.23	0.00	0.07	0.03	5.34	3.53	0.01
	Mothra	95.32	0.19	0.09	0.20	0.01	0.15	0.01	1.35	2.64	0.03
Equiv	IMSCU	90.32	0.49	0.06	0.24	0.00	0.07	0.04	5.28	3.49	0.01
	Mothra	95.35	0.19	0.09	0.21	0.01	0.15	0.01	1.34	2.62	0.03
Decomp	IMSCU	90.90	0.61	0.08	0.29	0.00	0.09	0.04	4.79	3.19	0.01
	Mothra	95.42	0.20	0.08	0.21	0.01	0.17	0.02	1.25	2.60	0.03
MutGen	IMSCU	91.27	0.52	0.06	0.25	0.00	0.07	0.04	4.65	3.13	0.01
	Mothra	94.93	0.22	0.10	0.22	0.01	0.16	0.02	1.17	3.15	0.02
TCEnt	IMSCU	90.66	0.56	0.10	0.26	0.00	0.07	0.04	5.00	3.31	0.01
	Mothra	95.44	0.23	0.09	0.20	0.01	0.15	0.01	1.29	2.54	0.03
Interp	IMSCU	91.42	0.58	0.12	0.23	0.00	0.06	0.04	4.53	3.01	0.01
	Mothra	95.55	0.24	0.12	0.22	0.01	0.18	0.02	1.12	2.51	0.03
Stats	IMSCU	90.72	0.47	0.10	0.23	0.00	0.08	0.04	4.94	3.40	0.01
	Mothra	95.52	0.19	0.09	0.19	0.01	0.16	0.01	1.18	2.62	0.03

**Table 3.** Comparison of Mothra and IMSCU.

One problem with Table 3 is that Mothra has a higher percentage of **independent** coupled module pairs than IMSCU, which is largely a result of the fact that Mothra has more files than IMSCU. If we ignore **independent** coupled modules, and define *coupled* module pairs to be any pair of modules that have some positive level of coupling, then we get a different, and perhaps more accurate view of how highly coupled the module pairs are. Figure 3 is a bar graph that compares Mothra and IMSCU by showing the percentage of coupled modules that exhibit each level of coupling.

For example, the top bar shows that of the IMSCU library modules that are coupled, about 1% are **call** coupled, over 50% are **non-local** coupled, and about 35% are **global** coupled, whereas Mothra library modules have almost no **call** coupling, about 30% are **non-local** coupled, and over 50% are **global** coupled.

Generally, Figure 3 shows that when modules do interact in some way, Mothra uses a much higher percentage of globals than non-local variable references.

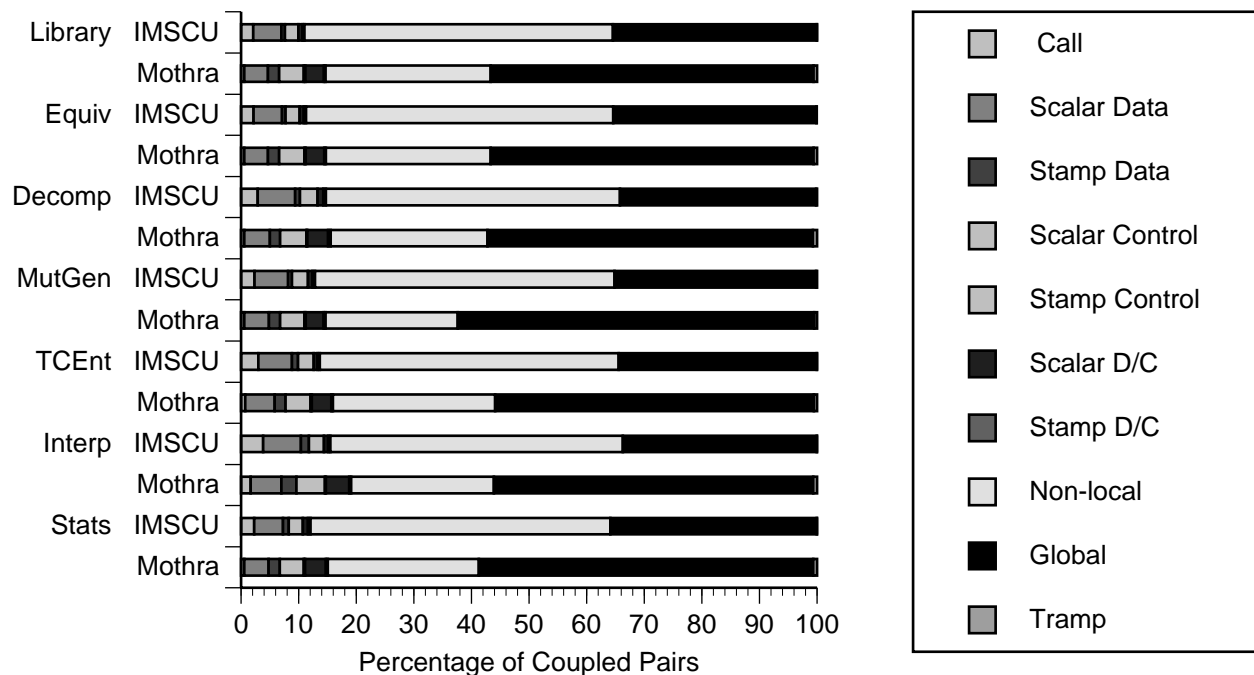


Figure 3. Mothra vs. IMSCU

The bar graph in Figure 4 shows the coupling measures for the Make program that were computed using Equation 2. It shows the number of coupled module pairs that exhibit the coupling measure for each coupling level. For example, there were 67 module pairs coupled at level 2 (*scalar data*) by one parameter, 30 pairs at level 2 with 2 parameters, etc. This graph shows that Make usually had only a few connections between coupled module pairs, and in all cases, there were more connections with one parameter or variable than multiple parameters or variables. This was true for all of the software we examined, and if true in general, might indicate that complexity measures that incorporate the number of connections, such as Fenton and Melton’s [4], provide more detail than necessary.

## 5 Future Work

Although module coupling is widely accepted as a way to evaluate the quality of design, its importance depends on whether coupling can be used effectively in the software production process. One use of module coupling is as a way to predict faults in the software. As mentioned in section 2, Troy and Zweben [15] studied the relationship of coupling to the number of faults found in programs, and found that module coupling was a good predictor of the number of faults in software. Selby and Basili [14] had a similar experiment based on the *clustering* measure proposed by Hutchens and Basili [6]. In the future, we plan to repeat such an experiment, using our automated means of determining coupling, and using our revised coupling levels.

As mentioned in section 3.3, our implementation does not include *external* coupling. The difficulty with determining *external* coupling is that most references to external media (such as files) are through file pointers, so which file is being accessed can only be determined at run time. One pessimistic approach would

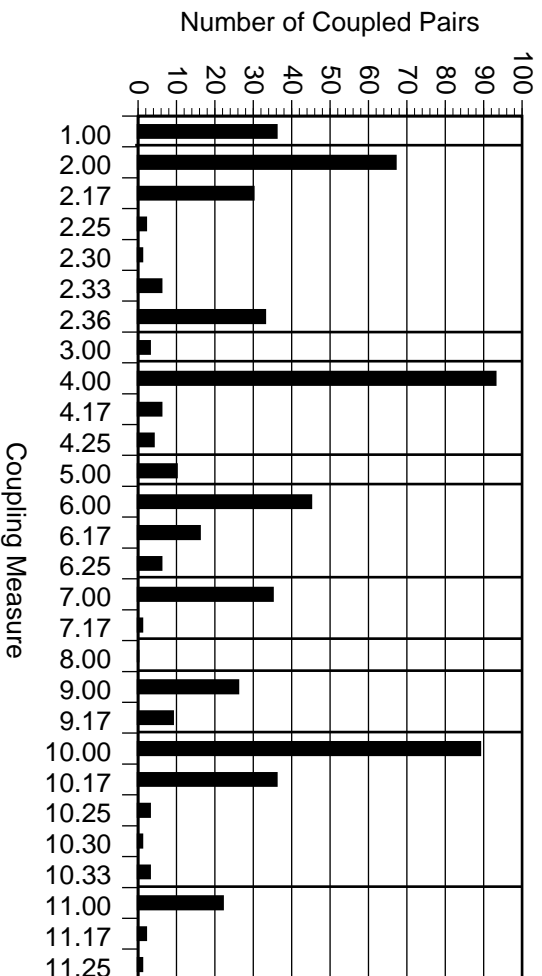


Figure 4. Coupling Measures for Make

be to consider all file accesses to be coupled externally. A more expensive, but more accurate, approach would be to use slicing techniques to determine which accesses can and cannot be to the same external file. We are currently exploring algorithms such as these for including **external** coupling into our implementation.

## 5.1 Coupling between Modules with More than One Procedure

Most of the previous research in coupling has defined a module as being one procedure. However, with the recent use of data abstraction, modular design, and object-oriented design, software designers are creating collections of procedures within one module that are highly coupled to each other but loosely coupled to procedures in other modules. Unfortunately, the current measures of coupling do not distinguish between procedures in one module and those in other modules. Thus, they may indicate that procedures in a module are highly coupled without considering that most of the coupling is limited by the boundaries of the module. However, high levels of coupling among the procedures in a module is desirable precisely because the coupling is so limited, and we suggest that new coupling measures need to take these kinds of designs into account.

To do this, we first measure the coupling **within** a module. Most multi-procedure modules contain procedures that operate on one or more of the same data structures, or parameters of the same type. Thus, we define the following three levels of coupling:

1. **Independent Coupling** - The procedures share no variables or types. For example, this module may contain utility procedures that are similar in function.
2. **Type Coupling** - The procedures access data structures that are of the same type. They may be passed as parameters or return values of functions.
3. **Structure Coupling** - The procedures access one or more non-local data structures that are defined within the scope of the module but that are not available to procedures in other modules.

Next, we measure the coupling between two modules by measuring all the couplings between procedure pairs in the modules, and then computing the coupling between the two modules as the maximum of the

procedure pair couplings. This computation is similar to determining coupling between pairs of single-procedure modules. In the future, we plan to implement this algorithm to determine coupling between multi-procedure modules.

## 6 Conclusions

In this paper, we have made five contributions to module coupling research. First, we have extended the coupling levels of Constantine, Yourdon, and Myers by adding `call coupling`, `stamp coupling`, `scalar data/control coupling` (indirect), and `stamp data/control coupling` (indirect), and by splitting `common coupling` into `non-local coupling` and `global coupling`. In some cases, these extensions reflect features of modern programming languages, and in others, they were necessary to fully quantify the coupling levels. Second, we have given precise, *prescriptive* definitions for each coupling level. These definitions are translated into the algorithms given in section 3.2. Third, we have used these algorithms to implement a powerful and practical automated software metric system that measures the levels of coupling between all module-pairs in a program. Fourth, we have presented coupling measures from several well-used C programs. During our experimentation, we were able to compare two similar systems (Mothra and IMSCU), and compute measures that match our pre-existing understandings of the systems; that IMSCU is more modular, and less coupled than Mothra. The fact that our system yields results that are expected gives us some confidence that module coupling is a reasonable measure of software design quality. Finally, we have suggested a way to take into account the restricted form of higher coupling that is a result of designs that are based on data abstraction.

The coupling levels that we offer provide a very fine grain resolution measure, and may be finer than necessary for some applications. Whether this level of precision is needed depends on how our pairwise module coupling measurement is to be used. In situations where our coupling levels are too detailed, Kafura and Henry's may be more useful, or ours could easily be summarized to result in a measure that has less resolution.

We plan experiments to determine the relationship between our coupling measures of a program/system and the number of faults (similar to what Troy and Zweben [15] and Basili and Selby [14] have done). We expect that the ability to automatically measure module coupling will allow programmers to use this important measurement in practical situations such as determining the maintainability of the system, the difficulty of regression testing when changes are made, and the reusability of modules.

## References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.
- [2] Constantine and Yourdon. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.
- [3] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.
- [4] N. Fenton and A. Melton. Deriving structurally based software measures. *The Journal of Systems and Software*, 12(3):177–886, July 1990.

- [5] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, October 1988.
- [6] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, August 1985.
- [7] P. Jalote. *An Integrated Approach to Software Engineering*. Springer-Verlag, New York NY, 1991.
- [8] D. Kafura and S. Henry. Software quality metrics based on interconnectivity. *The Journal of Systems and Software*, 2:121–131, 1981.
- [9] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software—Practice and Experience*, 21(7):686–718, July 1991.
- [10] G. Myers. *Reliable Software Through Composite Design*. Mason and Lipscomb Publishers, New York NY, 1974.
- [11] A. J. Offutt and S. D. Lee. IMSCU programmer’s reference manual. Technical report 91-121, Department of Computer Science, Clemson University, Clemson SC, 1991.
- [12] A. J. Offutt and R. H. Untch. Integrating research, reuse, and integration into software engineering courses. In *1992 SEI Conference on Software Engineering Education*, San Diego, California, October 1992.
- [13] M. Page-Jones. *The Practical Guide to Structured Systems Design*. YOURDON Press, New York, NY, 1980.
- [14] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, February 1991.
- [15] D. A. Troy and S. H. Zweben. Measuring the quality of structured designs. *The Journal of Systems and Software*, 2:112–120, 1981.
- [16] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
- [17] W. Wulf and M. Shaw. Global variables considered harmful. *Sigplan Notices*, 8(2):28–34, February 1973.