Metrics for Modularization Assessment of Scala and C# Systems

Basavaraju Muddu
Infosys Labs
Bangalore, India
mbraju@infosys.com

Allahbaksh Asadullah
Infosys Labs
Bangalore, India
allahbaksh_asadullah@infosys.com

Vasudev Bhat
Infosys Labs
Bangalore, India
vasudev_d@infosys.com

Srinivas Padmanabhuni
Infosys Labs
Bangalore, India
srinivas_p@infosys.com

***Abstract*—Modularization of a software system leads to software that is more understandable and maintainable. Hence it is important to assess the modularization quality of a given system. In this paper, we define metrics for quantifying the level of modularization in Scala and C# systems. We propose metrics for Scala systems, measuring modularization with respect to concepts like referential transparency, functional purity, first order functions etc., which are present in modern functional programming languages. We also propose modularity metrics for C# systems in addition to the Object Oriented metrics that are existing in the literature. We validated our metrics, by applying them to popular open-source Scala Systems - Lift, Play, Akka and C# systems - ProcessHacker and Cosmos.**

***Index Terms*—modularization, metrics, scala, functional programming, modularity, C#, modularity metrics.**

I. Introduction

Software systems evolve over a period of time. The systems go through bug-fixes and enhancements as part of maintenance, resulting in growth in size and complexity. This leads to higher time to fix bugs and affects the time to market ratio. The mantainability of such systems can be improved by having the right set of modules. A module is a logical collection of related files which can be modified and tested independently. A well modularized system will have low inter-module coupling and high intra-module cohesion. It helps the designers to add new modules easily.

To ensure good modularization, we need to measure and monitor the quality of modularization. Sarkar et al.[3] defined a set of modularity principles based on which the modularity can be measured through metrics. These metrics measure in different ways the interactions between the modules, and thus indicate the modular health of a system which the system maintainers can use to evaluate and refactor the system for better maintainability. The modularity metrics complement the conventional code metrics, which measure code properties, like complexity. Further work, by Sarkar et al. in [4], extended the research on modularity metrics to the Object Oriented paradigm, with an emphais on the Java programming language.

In this paper, we propose a set of modularity metrics for Scala and C# programming languages. We selected Scala because of its increasing popularity and its capabilities that go beyond the pure object oriented programming paradigm. As it supports object oriented programing, most of the existing metrics defined for Object Oriented systems are relevant for Scala. In the current work, we focus on its functional programming capabilities, like functional purity, first order functions, referential transparency, etc., when measuring the modularization quality. C# is an object oriented language, similar to Java and also supports few functional programming features. We propose additional modularity metrics for C# systems that consider these features.

In secton II, we briefly discuss the related work on modularization and modularity metrics. We define the proposed metrics for Scala, along with a brief review of notation used in the paper, in section III. Later, we explain additional metrics for C# in section IV. We discuss the approach we used for validating the metrics on open source software systems in section V, and later discuss results in section VI. We present threats to validity of our metrics in section VII, before concluding in section VIII.

II. Related Work on Metrics

While early work on software metrics focused on measures of code complexity[1] [2], the metrics for measuring modularization are based on cohesion and coupling. Briand et al.[6] formalized the notions of modularity, like module, system, coupling etc., and defined a framework for modularity metrics. Coppick and Cheatham [7] extended the code complexity metrics to Object Oriented (OO) systems. Subsequent works on OO metrics, by Brito et al.[8], Chidamber and Kemerer (CK) [9], Henderson et al.[10], focused on how a class's complexity and its linkage to other classes can be measured. Briand et al.[11] proposed a framework for characterizing metrics for OO systems.

Researchers have applied clustering approaches [17] to arrive at modules as clusters of programs. Automated modularization approaches have an objective of increasing a modularization quality measure. Mancoridis [12] used a metric, Modularization Quotient (MQ) based on function call dependencies between the modules, in their automatic modularization tool, Bunch. Shokoufandeh et al.[13] defined a Cluster Factor (CF) for each cluster as the ratio of a cluster's internal functional call dependencies to sum of both internal and external call dependencies of the cluster. Allen et al.[14] proposed metrics based on information theoretic approaches to measure the cohesion and coupling between software modules. Sarkar et al.[3][4] defined an API based approach for modularization

assessment, and proposed a set of metrics for C and Java systems. Abdeen et al.[19] proposed a set of modularity metrics for Java systems, measuring inter-package and intra-package dependencies, having considered a Java package as a module. Ryder [15] and van den Berg [16] defined software measures for functional programming systems from code complexity perspective.

Our work primarily focusses on measuring the impact on modularity due to the functional paradigm in Scala and C# systems. This work is an extension of our earlier contribution on metrics for functional programming systems[18].

III. Modularity Metrics For Scala

We define modularity metrics to account for the functional programming features of Scala, like first order function usage and referential transparency [5], as well as certain language-specific features. The metrics' values will be in the range 0 to 1. Any value close to 1 indicates the system is fine with respect to the concept measured by the metric for modular design, and any value close to zero suggests the system behaves poorly w.r.t. the corresponding concept and needs improvement. We use the following symbols in this paper:

- We express the software system by S, which consists of a set of modules $\wp = \{p_1, ..., p_N\}$ where each p_i is a system module and 1 $\leq$ i $\leq$ N.
- The set of all classes in system S is represented by C, and C(p) expresses the subset of the classes contained in module $p \in \wp$.
- M is used to represent the set of all methods or functions within the system. M(c) is the set of methods defined in class $c \in C$, and M(p) denotes the set of functions defined in module $p \in \wp$. Inversely, Mod(c) is the module in which class $c \in C$ resides, while Mod(m) is the module in which the function $m \in M$ resides.
- NC(m_1,m_2) is the number of calls to m_2 from m_1.
- FOC(m_1,m_2) indicates the existence of m_2 as a parameter in a function call, within m_1, while NFOC(m_1,m_2) shows the number of such calls.
- The operator InUse(m) analyzes if a method is called during program execution (belongs to the execution tree).
- Pure(m)=1 means m is a pure function.

A. Referential Transparency Metrics

A key concept in functional programming (FP) is the referential transparency, that is, a same function, irrespective of when or how it is called, should yield the same results if the same inputs are provided. This is referred to as a pure function, meaning it has no side effects, other than returning a result. The use of this kind of functions is extremely useful for code isolation, as these functions can be used indiscriminately by anyone, without any side effects. We define the metric, Referential Transparency Index (RTI) as the proportion of all the defined methods which can be considered pure functions. If the modularization scheme is well applied, only a very small part of the methods would be non-pure. The RTI for the system is defined as follows:

$$RTI = \frac{|\{m \in M | Pure(m) \wedge InUse(m)\}|}{|\{m \in M | InUse(m)\}|} \quad (1)$$

Referential transparency becomes a critical measure when the functions are being called from one module to another, since changes in the caller module may affect the called module, which results in programmers re-testing their code after every change. In the Intermodule Referential Transparency Index (IRTI) we measure the proportion of the calls in a certain module to methods in other modules which are pure over all the intermodule calls:

$$IRTI(p) = \frac{\sum_{m \in M(p)} \sum_{m' \in \{M - M(p)\}} NC(m, m') \times Pure(m')}{\sum_{m \in M(p)} \sum_{m' \in \{M - M(p)\}} NC(m, m')} \quad (2)$$

In the numerator, the 'pure' operator will have a zero value if it is non pure, 1 otherwise. Multiplying by it will exclude non-pure ones.

To extend the metric to the entire system, we take the average of IRTI(p):

$$IRTI(S) = \frac{1}{|\wp|} \sum_{p \in \wp} IRTI(p) \quad (3)$$

B. First Order Function Metrics

In functional programming the use of first-order functions (that is, functions passed as arguments or returned by other functions) is a standard practice. It allows the reuse of parts of code as building blocks by separating the way to treat a data structure from the actual task to be performed on it. This is clearly an advantage w.r.t modularization, but it may cause more problems than benefits, if used improperly across the modules. If we use methods that reside in different modules as first-order functions, we are creating a dependency between them. This problem is accounted for by the metric Intermodule First Order Functions Index (IFOI). The extent of this form of dependency issues can be measured in three different ways.

The metric $IFOI_1$ determines the proportion of the methods in a certain module which make first order function calls to methods in another module:

$$IFOI_1(p) = 1 - \frac{|\{m \in M(p) | \exists_{m' \in M} FOC(m, m') \wedge Mod(m') \neq p\}|}{|m \in M(p) | \exists_{m' \in M} FOC(m, m')|} \quad (4)$$

The numerator counts the number of methods in a module which include at least one call to a function that uses a function defined in another module as its parameter, whereas, the denominator is the number of methods in the module, having calls to functions with a function as a parameter. $IFOI_2$ measures the proportion of out-side first-order-calls used by the current module, by counting all the methods

defined outside the module that are called in a first-order way by methods in the current module:

$$IFOI_2(p) = 1 - \frac{|\{m' \in M | \exists_{m \in M(p)} FOC(m, m') \wedge Mod(m') \neq p\}|}{|m \in \{M - M(p)\} | \exists_{m' \in M} FOC(m, m')|} \quad (5)$$

Here the numerator shows all the methods not belonging to module p, which are called by methods m belonging to module p, in a first order way, to be normalized by the amount of methods that are outside module p.

$IFOI_3$ measures the proportion of the other modules whose methods are called upon in a first order way by current module's methods, that is,

$$IFOI_3(p) = 1 - \frac{|\{p' \in \wp | \exists_{m' \in M(p')} FOC(m, m') \wedge p' \neq p\}|}{|\wp| - 1} \quad (6)$$

Finally, we define the metric IFOI as the minimum of the three measures:

$$IFOI(p) = min(IFOI_1, IFOI_2, IFOI_3) \quad (7)$$

$$IFOI(S) = \frac{1}{|\wp|} \sum_{p \in \wp} IFOI(p) \quad (8)$$

However, when taking into account the pure function concepts reviewed in earlier sections, we want to add that a desirable quality for a function being passed as an argument to another function, for the purpose of proper modularization, is that the function being passed, should not have any side effects when run, leaving the non-functional tasks to the superior function. For this, we define the Pure First Order Function Index (PFOI) at system level to check the extent to which that principle is respected. The definition is as follows:

$$PFOI = \frac{\sum_{m \in M} \sum_{m' \in M} NFOC(m, m') \times Pure(m')}{\sum_{m \in M} \sum_{m' \in M} NFOC(m, m')} \quad (9)$$

$PFOI$ measures the proportion of all the first order calls which are to pure methods. An undesirable condition is to have first order calls happening from one module to another, and with a non-pure function being called. This is a combination of the undesirable traits measured by IFOI and PFOI, but its value cannot be derived out of them, since it is the intersection of both sets, which might perfectly well be mutually exclusive. This metric is called the Intermodule Pure First Order Function Index (IPFOI) and is defined as the ratio between the first order calls to pure methods residing in another module over all the first order calls done:

$$IPFOI(p) = \frac{\sum_{m \in M(p)} \sum_{m' \in \{M - M(p)\}} NFOC(m, m') * Pure(m')}{\sum_{m \in M(p)} \sum_{m' \in \{M - M(p)\}} NFOC(m, m')} \quad (10)$$

$$IPFOI(S) = \frac{1}{|\wp|} \sum_{p \in \wp} IPFOI(p) \quad (11)$$

C. Inheritance based Metrics

The inheritance-based intermodule coupling metric, IC, defined by [4], is modified to account for the multiple inheritance provided by Scala through *mixins*. Mixins allow the programmers to reuse the delta of a class definition, i.e., all new definitions that are not inherited, when inheriting from multiple classes. We add to the IC1, IC2 and IC3 defined by [4], one Mixin Inheritance Intermodule Coupling (MiIC), and calculate the MIC as the minimum of the four.

A module may have classes derived from classes in other modules through direct inheritance or mixins. To indicate the degradation in the modularization quality due to multiple inheritance across modules, we define the power range R(c), the inheritance range of a class c, as the number of different modules from which it inherits, i.e., the modules of the classes the current module's classes inherit from:

$$R(c) = |\{p' \in \wp | \exists_{d \in C(p')} Chld(c, d) \wedge Mod(c) \neq p'\}| \quad (12)$$

Here, $Chld(c, d)$ means that class c is a child of class d. We extend the notation, calling the range of a module R(p) as the maximum of the ranges R(c) of all the classes in the module:

$$R(p) = max(R(c) | c \in C(p)) \quad (13)$$

We define a metric MiIC for a module, as the proportion of the classes in a module which inherit from classes in another module, using as exponent the maximum range($R(p)$) of the module to penalize the multimodule coupling. The MIC(p) is the minimum of IC1, IC2, IC3 and MiIC.

$$MiIC(p) = (1 - \frac{|\{c \in C(p) | \exists_{d \in C} Chld(c, d) \wedge Mod(d) \neq p\}|}{|C(p)|})^{R(p)} \quad (14)$$

$$MIC(p) = min(IC_1, IC_2, IC3, MiIC) \quad (15)$$

$$MIC(S) = \frac{1}{|\wp|} \sum_{p \in \wp} MIC(p) \quad (16)$$

IV. Additional Metrics for C#

A. Partial Types Metric

In large software systems, it is often desirable to split a Class definition over many source files. This is not possible in languages like Java, while C# facilitates the same via partial types. To ensure proper modularization of the code, it is important to minimize the number of types defined across the modules. Splitting a class across different modules is not desirable for a well modularized system, so we define an Intermodule Type Definition Index (ITDI) for a given module p, as the ratio between the classes with partial definitions in both module p and a different module p', over the amount of partial classes defined in p. Here, we use $Part(c, p)$ to indicate partial definition of class exists in module p.

$$ITDI(p) = 1 - \frac{|\{c \in C(p) | \exists_{p' \in \wp} Part(c, p') \wedge Part(c, p) \wedge p' \neq p\}|}{|\{c \in C(p) \wedge Part(c, p)\}|} \quad (17)$$

TABLE I
METRICS SUMMARY

Scala Metrics		
#	**Code**	**Name**
1	RTI	Referential Transparency Index
2	IRTI	Intermodule Referential Transparency Index
3	IFOI	Intermodule First Order Index
4	PFOI	Pure First Order function Index
5	IPFOI	Intermodule Pure First Order function Index
6	MIC	Multiple Inheritance Coupling
C# Metrics		
1	ITDI	Intermodule Type Definition Index
2	IEMU	Intermodule Extension Method Usage Index
3	DUI	Delegate Usage Index

$$ITDI(S) = \frac{1}{|\wp|}\sum_{p\in\wp} ITDI(p) \tag{18}$$

B. Extension Methods Metric

Extension methods extend a class by defining separate methods without modifying or inheriting the class. This is the only way to extend a sealed class in C#, which cannot be inherited from. This introduces a problem w.r.t modularization, when the extension method and the extended class reside in different modules. The Intermodule Extension Method Usage (IEMU) metric, which has two different parts, addresses this problem. We use $Ext(m, c)$ to imply that method m extends class c. We consider the proportion of the methods of a module p which extend classes outside p ($IEMU_1$):

$$IEMU_1(p) = 1 - \frac{|\{m \in M(p) | \exists_{c\in C} Ext(m, c) \wedge Mod(c) \neq p\}|}{|\{m \in M(p) | \exists_{c\in C} Ext(m, c)\}|} \tag{19}$$

Now, we define the ratio between the classes in module p that are extended by external methods over the total amount of extended classes in p as $IEMU_2$:

$$IEMU_2(p) = 1 - \frac{|\{c \in C(p) | \exists_{m\in M} Ext(m, c) \wedge Mod(m) \neq p\}|}{|\{c \in C(p) | \exists_{m\in M} Ext(m, c)\}|} \tag{20}$$

Finally, we obtain the IEMU metric for module p as the average of the above two:

$$IEMU(p) = \frac{IEMU_1 + IEMU_2}{2} \tag{21}$$

$$IEMU(S) = \frac{1}{|\wp|}\sum_{p\in\wp} IEMU(p) \tag{22}$$

C. Delegates

Similar to the function pointers used in C/C++, 'Delegate' in C# is a function prototype, having arguments and return type defined. It is instantiated first, pointing to an existing function, and later can be passed to another function or executed like any other function. While delegates are useful for functional programming and first order functions, they also help in improving modularization. Delegates can be used to point to methods external to current module, and thus act as a decoupling mechanism between the module. We propose a metric to measure the intermodule traffic through delegates, called Delegate Usage Index (DUI). It is defined as proportion of the delegates' usage for external calls w.r.t. all external calls from a module, both direct and through delegates.

Since we are performing static code analysis, we consider 'external call' if a delegate is initialized to an external function and called in a function of current module. In the formulae below, $Del(f)$ refers to a delegate pointing to a method f. Table I summarizes all the metrics defined for Scala and C# systems.

$$DUI(p) = \frac{\sum_{m_j\in M(p)} \sum_{f_k\in\{M-M(p)\}} NC(m_j, Del(f_k))}{\sum_{m_j\in M(p)} \sum_{f_k\in\{M-M(p)\}} (NC(m_j, Del(f_k)) + NC(m_j, f_k))} \tag{23}$$

$$DUI(S) = \frac{1}{|\wp|}\sum_{p\in\wp} DUI(p) \tag{24}$$

V. METRICS VALIDATION

We developed a system to validate the set of metrics. This system extracted the metadata from the given Scala and C# systems' source code and stored it in a database. We devised an experimental approach to assign modules and then calculated the metric values on the extracted data set.

A. Modularization

For any system, the concept of module boundaries is slightly open to interpretation, since only the architect of the software system knows the actual modules. We use heuristics to arrive at modules of a given input system in our experiments, as we could not obtain the real modules of the systems. One heuristic is the Direct-Depth approach where we fix a depth from the project's source folder and consider each sub-tree branching from that depth as a module. However because of the directory conventions used by the developers, modules can be too broad for some directories and too small for others. Hence, we used another heuristic, called, Modularizing by Inverse-Depth.

In the Inverse-Depth approach, the maximum depth is measured for each path. A module is defined by ascending a given number of directories from the maximum depth and assigning every file under it to the module. Depth 0 means all the leaves of the directory structure become a module each. By manually examining directory structures from open source projects we found that inverse depth 1 tends to be a close approximation to a logical modularization, i.e., one level above the leaves.

B. System Design

The system that we designed for Scala relies on the scalac compiler for source code parsing. We developed a plugin to the scalac compiler to visit the AST (Abstract Syntax Tree) created by the compiler. This extracts code metadata like classes, methods, parent-child relationships, method calls, etc., and stores it in a database (MySQL) (see Figure 1).

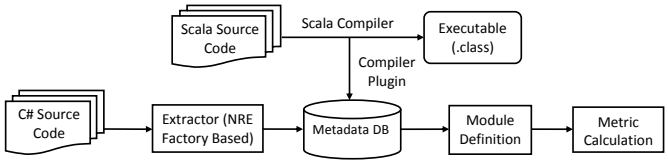


Fig. 1. Block Diagram of Designed System

The system for C# extracts the metadata from the source code, leveraging the open source library of NREFactory for parsing the C# source code and stores the metadata in the database.

Another system extracts the data from the database and performs the modularization according to the inverse depth approach and calculates the metrics. Figure 1 depicts the schematic diagram of the entire system.

It is important to note that the actual assessment of functional purity is a complex subject. For this work we have taken the path of annotating certain external libraries as pure and considering every other external method non-pure.

C. Analyzed Systems

Since the quality of modularization is particularly relevant in large software systems, we reviewed a few open source softwares developed in Scala, consisting of 200 to 400 files. To the best of our knowledge, these are the largest open source Scala softwares. It is important to point out that Scala code is much less verbose than its Java counterpart. We chose to evaluate Lift, which is a web applications framework, and Akka, which provides runtime functionality for concurrent programming, and Play 2.0 web framework.

For evaluating the proposed C# metrics, we chose a few open source C# systems, whose source code consisted of atleast 500 source files. We experimented with ProcessHacker, a free open-source utility that displays all the processes running on a computer in a compact view and Cosmos, an Open Source Managed Operating System written in C#.

We analyzed two versions of Lift, three versions of ProcessHacker and two versions of Cosmos to study the metrics as the software evolved from one version to next. We also tested the metrics for random modularization.

VI. Results and Discussion

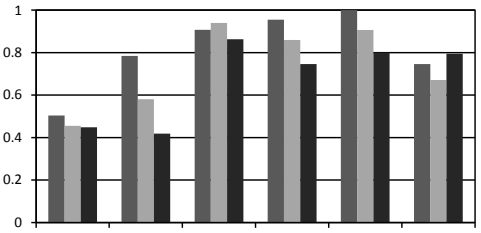
A. Comparison of Resulting Metrics of Different Software

The open source softwares that we chose for Scala, consisted of more than 20 modules (see Table II), and their metrics' results are shown in Figure 2.

The results show that Akka has the best modularization quality, by having the best results for most of the metrics. This is logical, because it offers a set of flexible tools for concurrent application development, meaning its very functionality relies on the usability of its functions by other programmers. Hence, the code needs to be properly modularized. The perfect score in the IPFOI is noteworthy, and shows that even though there were first order calls to non-pure methods and intermodule first order calls, none of the latter was to non-pure methods.

TABLE II
External Software Data Of Scala Systems

Software	Version	Source Files	Modules
Lift	2.3-final	401	29
Lift	2.5-final	423	29
Play	2.0.1	250	31
Akka	1.1-M1	160	30



	RTI	IRTI	IFOI	PFOI	IPFOI	MIC
Akka 2.1	0.5037	0.7844	0.907	0.9549	1	0.746
Lift 2.5	0.4552	0.5803	0.9393	0.8597	0.9062	0.6713
Play 2.0	0.448	0.4188	0.8629	0.746	0.8	0.7944

Fig. 2. Metric results for different Scala systems

The lite use of first order calls (the 0.9%) is also responsible for the good performance in IFOI, PFOI and IPFOI.

Lift, on the other hand, provides web services with a loose coupling to the programmers code, making proper modularization a less pressing concern for the system. Its results are still good, especially considering it uses first order calls in more than 4% of all the calls made. The lowest performance is seen in the MIC index, and is explained by the heavy inheritance coupling created by all the traits that a class needs to inherit to access web services of the lower layer, which will usually be defined in a different module.

The Play framework does not show as good results as the others, except for the MIC. This is associated with the relatively recent migration to Scala. The good performance in MIC is explained by the very lite use of multiple inheritance and we relate this to the Java origins of the system.

For the validation of C# metrics, we chose three stable versions of Process Hacker, 1.5, 1.8 and 1.9, which had undergone significant changes and two stable versions of Cosmos (Milestone 4 and Milestone 5). Both the projects consist of more than 600 files and more than 30 modules (see Table III), using inverse depth 1 modularization approach. Figure 3 shows the metrics results for these systems.

We observed that the modularization quality of Cosmos was better than ProcessHacker. Cosmos produced the the best metrics' values when compared to ProcessHacker. This could be attributed to the fact that Cosmos is an open source operating system written in C# and is one of the largest systems, used by the C# open source community and we expect it to be well modularized.

B. Metrics Evolution with Later Versions

For the Lift framework, we applied our metrics on two different versions that include no major breakthroughs in

TABLE III
EXTERNAL SOFTWARE DATA OF C# SYSTEMS

Software	Version	Source Files	Modules
ProcessHacker	1.5	526	39
ProcessHacker	1.8	624	41
ProcessHacker	1.9	178	14
Cosmos	Milestone 4	1459	64
Cosmos	Milestone 5	1471	60

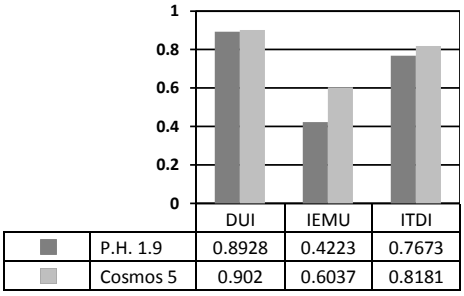


Fig. 3. Metric results for C# systems, P.H. (ProcessHacker) & Cosmos

code capability, under the assumption that the evolution does include an improvement in modularization. It is important to point out that it is vital for a framework to improve modularity since it is used by different programmers for different objectives.

When calculating our metrics on two different versions of the same software (Lift 2.3 final and Lift 2.5 final) we found that all metrics either improve or remain same, as depicted in Table IV. As they are close versions, released within less than a year difference, the metric changes are expected to be small. This is evident from the fact that there was no major change in the system as observed by mining the version control system.

The highest increase happens in the IRTI, with an improvement of 33.8%, partly related to the decrease in the number of impure functions. It is noteworthy that the IRTI changed by 33.8% against only 5% of improvement in RTI, which indicates that most of the functions that have become pure (or been replaced by pure methods) are the ones called between modules. This strongly supports the principles on which we based our IRTI metric.

There is a small improvement of 0.3% in the PFOI showing that some of the first order calls have been switched to pure methods, which also accounts for an improvement of 0.22% in the IPFOI. The increment in first order calls metric is due to the fact that most of such calls are located in some of the core functionalities of the framework, and therefore received few modifications during the 2.5 version. Finally, the MIC improvement of 1.34% shows that fewer intermodule inheritance relations were present in the later version. A closer analysis showed that the main factor responsible for this improvement in the MIC is relocation of some traits widely used in a module different than the one in which it was defined.

TABLE IV
METRICS ACROSS VERSIONS FOR LIFT FRAMEWORK

	RTI	IRTI	IFOI	PFOI	IPFOI	MIC
Lift 2.3	0.4335	0.4337	0.9372	0.8569	0.9042	0.6624
Lift 2.5	0.4552	0.5803	0.9393	0.8597	0.9062	0.6713

TABLE V
METRICS ACROSS VERSIONS FOR C# SYSTEMS

	DUI	IEMU	ITDI
ProcessHacker-1.5	0.71	0.379	0.7073
ProcessHacker-1.8	0.7314	0.3843	0.7214
ProcessHacker-1.9	0.8928	0.4223	0.7673
Cosmos-M 4	0.802	0.4089	0.8181
Cosmos-M 5	0.902	0.6037	0.8181

For evaluation of C# metrics across versions, we considered two versions of Cosmos and three versions of ProcessHacker for studying the changes in metrics as the software evolved. Table V depicts the results. In both the systems, the metrics changed as the system migrated from one version to the other. With ProcessHacker, the metrics improved by a small percentage of 2.1% on average when moving from version 1.5 to 1.8. But the metrics improved significantly by 12% on average in the version 1.9. In version 1.9, C# code was better modularized, and few core parts of the code were rewritten in C. The DUI metric increased by 22% moving from version 1.8 to 1.9. This indicates that the number of methods accessed across modules through delegates increased in version 1.9, which is an important aspect of a well modularized system.

Cosmos showed an increase in the values of DUI and IEMU metrics while migrating from Milestone 4 to Milestone 5. A significant increase of 47.64% was obtained in the values of IEMU which indicates careful usage of extension methods for extending classes outside the module. ITDI remained the same in both the versions of Cosmos since the number of partial classes going from Milestone 4 to Milestone 5 remained unchanged, although new classes were created. This supports one of the principles on which we based the ITDI metric, that the number of partial classes across modules should be minimized as the software evolves.

C. Inverse Depth v/s Random Modularization

We validated our metrics by comparing the results obtained from inverse depth approach, with the results obtained by randomly assigning files to different modules. To keep the situation measurement unbiased, we distributed files randomly in the same n number of modules obtained by the inverse depth modularization. This test was applied on Lift 2.5 web framework for Scala metrics and Cosmos Milestone 5 project for C# metrics. The results are shown in Table VI and Table VII respectively.

The first observation in these results is that RTI and PFOI metrics remain unchanged. Although both metrics give us information about the general modularization quality of certain software, they do not take into account the module in which entities reside, but the way they behave. Therefore, reassigning files to different modules should not change them.

TABLE VI
METRICS DEGRADATION WITH RANDOM MODULARIZATION

	Lift 2.5					
	RTI	**IRTI**	**IFOI**	**PFOI**	**IPFOI**	**MIC**
Rand	0.4552	0.4967	0.7494	0.8597	0.5239	0.5723
Dir	0.4552	0.5803	0.9393	0.8597	0.9062	0.6713

TABLE VII
METRICS DEGRADATION WITH RANDOM MODULARIZATION

	Cosmos Milestone 5		
	DUI	**IEMU**	**ITDI**
Rand	0.565	0.5142	0.3567
Dir	0.902	0.6037	0.8181

Major changes can also be observed in the IRTI, IFOI and IPFOI, which is consistent with the importance given by these metrics to the intermodule calls. We have stated earlier that the intermodule calls to non-pure functions (IRTI) and intermodule first order calls (IFOI) are not desirable. Also, the combination of intermodule first order calls to non-pure methods (IPFOI) degrade the modularization quality. In line with this concept, a bad (or random) distribution of files between modules will clearly result in a poor IPFOI value. For MIC metric, since class inheritance relationships are to be kept intramodule, a randomized modularization ruins the designer's effort to stay along these lines and hence shows a significant decrease in the value of the metric.

We carried out a similar validation on Cosmos Milestone 5 for C# metrics and observed that the metrics degraded when randomized modularization was applied. The IEMU and ITDI metrics indicate that the partial types and extension methods are not localized to individual modules when random modularization is applied. A bad modularization will result in the degradation of the values of DUI, IEMU and ITDI.

VII. THREATS TO VALIDITY

The proposed metrics are validated on open source software systems and are yet to be tested on large scale enterprise software, but the evidence so far obtained clearly validates the assumptions on which the metrics were based. The modules created during the experiments are not based on developer or architect inputs, who would have a better understanding of the system and the testing against such a system will further strengthen the validity of the metrics.

VIII. CONCLUSION

We have proposed a set of metrics for measuring the modularization quality of software systems developed in Scala and C#, that act as an addition to the Object Oriented metrics existing in the literature. It is important to provide a way for programmers to assess the extent to which they adhere to good functional programming practices in a well modularized system, for example, by maintaining functional purity as much as possible. Our contribution to the inheritance based metrics is to take into account the multiple inheritance capabilities offered by Scala and generate a metric that penalizes the coupling of multiple modules through inheritance and trait implementation. We proposed few additional metrics for C#, accounting for the impact of usage of partial types, extension methods and delegates on modularization.

The experiments we have done have provided encouraging evidence in favor of the set of metrics, showing that they behave as expected. The assessment of different software systems has shown values that are in accordance with the comparison of their modularization qualities, while an analysis of different versions of a system has proven that the metrics reflected small improvements on modularization done in the version evolution. The proposed modularity metrics measure the impact of functional programming features on the modularization quality of a system. We would like to extend this work and the guiding principles to other functional programming languages like F# and Clojure.

REFERENCES

[1] M.H. Halstead, "Elements of Software Science, Operating and Programming Systems Series", vol. 7, Elsevier, 1977.

[2] T.J. McCabe and A.H. Watson, "Software Complexity, Crosstalk, J. Defense Software Eng., vol. 7, no. 12, pp. 5-9, Dec. 1994.

[3] S. Sarkar, G.M. Rama and A.C. Kak, "API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization," *IEEE Trans. Software Eng.*, vol. 33, pp. 13-42, 2007

[4] S. Sarkar, A.C. Kak and G.M. Rama, "Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software," *IEEE Trans. Software Eng.*, vol. 34, pp. 700-720, 2008

[5] J. Hughes, "Why Functional Programming Matters," Technical Report 16, Programming Methodology Group, University of Goteborg, 1984.

[6] L.C. Briand, S. Morasca, and V.R. Basili, "Property-Based Software Engineering Measurement," IEEE Trans. Software Eng., vol. 22, no. 1, pp. 68-85, Jan. 1996.

[7] C.J. Coppick and T.J. Cheatham, "Software Metrics for Object- Oriented Systems," Proc. ACM Ann. Computer Science Conf., pp. 317-322, 1992.

[8] F. Brito e Abreu and R. Carapuca, Candidate Metrics for Object- Oriented Software within a Taxonomy Framework," J. Systems and Software, vol. 26, pp. 87-96, 1994.

[9] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, 1994.

[10] B. Henderson-Sellers, "Object-Oriented Metrics: Measures of Complexity." Prentice Hall, 1996.

[11] L.C. Briand, J.W. Daly, and J.K. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," IEEE Trans. Software Eng., vol. 25, no. 1, pp. 91-121, 1999.

[12] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code," Proc. Sixth Intl Workshop Program Comprehension (IWPC 98), pp. 45-52, 1998.

[13] A. Shokoufandeh, S. Mancoridis, T. Denton, and M. Maycock, "Spectral and Meta-Heuristic Algorithms for Software Clustering," J. System and Software, vol. 77, no. 3, pp. 213-223, Sept. 2005.

[14] E.B. Allen, T.M. Khoshgoftaar and Y. Chen, "Measuring Coupling and Cohesion of Software Modules: An Information Theory Approach," in Metrics '01: Proc. 7th Int. Symp. on Software Metrics, IEEE Computer Society, p. 124, 2001

[15] C. Ryder, "Software Measurement for Functional Programming," PhD thesis, Computing Lab, University of Kent, Canterbury, UK 2004.

[16] K.G. van den Berg, and P.M. van den Broek, "Static analysis of functional programs, Information and Software Technology," 37(4), pp. 213-224, 1995.

[17] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization." Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on. IEEE, 1997.

[18] M. N. Gubitosi, Basavaraju M., A. M. Asadullah. "Metrics for Measuring the Quality of Modularization of Scala Systems," Proc. 19th Asia Pacific Software Engineering Conference, vol. 2, pp. 9-16, 2012.

[19] H. Abdeen, S. Ducasse, and H. Sahraoui. "Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software." 18th Working Conference on Reverse Engineering(WCRE), 2011.