Z Formal Framework for Syntax-Based Module Level Software Metrics

Raouf Alomainy and Wei Li
Computer Science Department, University of Alabama in Huntsville, Huntsville, AL 35899
{ralomain, wli}@cs.uah.edu

Abstract

This paper introduces a framework to formalize module-level structural metrics that quantify inter-module dependencies in object-oriented software systems. We used a formal framework based on the Big Bang Graph (BBG) modelling and the formal Z specification language to formalize a modularization-based software metric as an example to demonstrate how the framework works. We have developed the Design State Space toolkit (DSS) to extract the state space represented in the formalized object-oriented model.

Keywords: Module-level structural metrics, Z formal language; design state space; software metrics tools;

Introduction

Modularization, as a feature of object-oriented programming paradigm, addresses the need for decoupled modules, each with a well-defined and advertised functionality, in order to achieve code reuse [5], separation of concerns [4], and minimize overlapping and unnecessary coupling among different modules [3]. Well-structured and modular programs are less costly to maintain than unstructured monolithic ones [2].

This paper focuses on structural metrics: quantitative measures of programs that are based on syntactic structures. Such metrics have been used to measure object-oriented programs to predict system properties such as class error probability and maintainability. An example of class-level structural metric is the Number of Children (NOC) metric [4] to calculate the number of immediate subclasses subordinated to a class in the class hierarchy in order to measure how many subclasses are going to inherit the methods of the parent class. An example of module-level structural metric is the Module Size Uniformity Index (MSUI) to assess if all modules are roughly equal in size [10]. The deviation from the module-size uniformity would generally be an indicative of a poor modularization.

Module-Level Structural Metrics

In building large systems, there is a consensus on the need for well-defined and structured architecture that is based on modularized software components, in particular for large-scale software system [13]. However, a significant body of previous research work has often considered a module and a class to be synonymous concepts. Thus, most of the research is focused on class-level metrics and there is a lack of empirical studies on module-level metrics and their influence on the disorganization that exist today in many commercial systems [10].

In this paper, we refer to a module as a collection of classes that are grouped together to serve a purpose in a large object-oriented system.

Several common software practices contribute to the undermining of the original purpose of modularization. Some examples of such errors: a class extending another class from a different module, a class in one module is integrating, through instantiation, with a class in another module, either as an attribute or as a formal parameter in a method definition, the messaging between methods that belong to classes of different modules for local purpose functionality, etc [10]. Module-level metrics are meant to gauge well or poorly the modularity has been constructed.

In this paper, we propose a formal framework, based on the Z formal language [12], that links structural module metrics to design features in order to aid the precise definition and better understanding of module-level metrics. Z has the ability to prove properties in its specification to formally validate the metric [7].

Sakar and colleagues proposed module metrics that characterize large object-oriented software systems with regards to the inter-module dependencies created by associations, inheritance, and method invocations [11]. In the
remainder of this paper, we present the formalization of one of these metrics using our proposed approach.

**Formal Modelling of Object-Oriented Design State Space**

The formal framework is based on a new graph modelling technique, Big Bang Graph (BBG), for OO programs and the formalization of the graph model using the Z formal language. The formalized BBG model is the *design state space* of the program.

The distinguished feature of BBG is its simplicity: it uses only one graph to represent the entire design state space that we are interested in modelling structural metrics.

BBG is a colored, directed and connected graph modelled as BBG = (E, V) where V is the set of colored vertices and E is the set of colored and directed edges. Vertices in BBG represent design entities such as class, object, object reference and method. Edges represent the primitive associations of the entities. BBG aims to model a program with the primitive (atomic) design features. A design feature is *primitive* or *atomic* if it cannot be further divided into other design features. For example, *object read* is a primitive design feature whereas *object access* is not because it consists of two primitive design features: *object read* and *object write*. For more details on the BBG modelling, please refer to [8].

**An Example**

We use a simple module-based Java program (Listing 1) to illustrate how the formal framework works. In Java, we equate a module to a package. The sample program consists of two Java packages: A and B, each with three Java classes. Package A consists of classes A1.java, A2.java, and A3.java. Package B consists of classes B1.java, B2.java, and B3.java.

```java
package A;
import java.io.*;
import java.util.*;

public class A1 {
    private Vector APrivateMethods;
    private Vector APublicMethods;
    public A1() {
    }
    public void MA11() { B1 obj_b1 = new B1(); obj_b1.MB11(); }
    public void MA12() { B2 obj_b2 = new B2(); obj_b2.MB21(); }
    public void MA13() { B2 obj_b2 = new B2(); obj_b2.MB21(); }
    public void MA14(String para1, Integer para2, Float para3) {
    } // end of class A1
}

package A2;
import B.*;
package A;

public class A2 {
    private Vector APrivateMethods;
    private Vector APublicMethods;
    public A2() {
    }
    public void MA21() { A1 obj_a1 = new A1(); obj_a1.MA12(); }
    public void MA22() {
    }
    public void MA23() {
    }
    public void MA24(String para1, Integer para2, Float para3) {
    } // end of class A2
}

package A3;
import B.*;
package A;

public class A3 {
    private Vector APrivateMethods;
    private Vector APublicMethods;
    public A3() {
    }
    public void MA31() { A2 obj_a2 = new A2(); obj_a2.MA21(); }
    public void MA32() {
    }
    public void MA33() {
    }
    public void MA34(String para1, Integer para2, Float para3) {
    } // end of class A3
}

package B;
import java.io.;
import java.io.io.;
import java.*, A.*;

public class B1 {
    public void MB11() {
    }
    public void MB12() {
    }
    public void MB13() { A3 obj_a3 = new A3(); obj_a3.MA31(); }
    public void MB14(String para1, Integer para2, Float para3) {
    } // end of class B1
}

package B2;
import A.*;
package B;

public class B2 {
    public void MB21() {
    }
    public void MB22() {
    }
    public void MB23() {
    }
    public void MB24(String para1, Integer para2, Float para3) {
    } // end of class B2
}

package B3;
import A.*;
package B;

public class B3 {
    public void MB31() {
    }
    public void MB32() {
    }
    public void MB33() { A2 obj_a2 = new A2(); obj_a2.MA21(); }
    public void MB34(String para1, Integer para2, Float para3) {
    } // end of class B3
}
```

Listing 1: Java source code for classes in packages A and B

Table 1 shows the generated BBG relation sets that will be used as the base types to support the
Z formalization of the module-level metrics. For more details on the complete list of BBG relations sets, please refer to [8].

Once the BBG relation sets are created from the Java example to represent the design space, the design can be analyzed formally. Syntax-based software metrics (structural metrics) can be defined and extracted from these sets, as shown in the next section using the Z formal language.

Table 1: Generated BBG relation sets

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Relation semantics</th>
<th>BBG Relation Set pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Class</td>
<td>Private instance</td>
<td>[(A), (A2), (A3), (B1), (B2)]</td>
</tr>
<tr>
<td>Class</td>
<td>Method</td>
<td>Private instance</td>
<td>[(A), (A1), (A2), (A3), (B1), (B2)]</td>
</tr>
<tr>
<td>Class</td>
<td>Method</td>
<td>Protected instance</td>
<td>[(A), (A1), (A2), (A3), (B1), (B2)]</td>
</tr>
<tr>
<td>Class</td>
<td>Method</td>
<td>Public instance</td>
<td>[(A), (A1), (A2), (A3), (B1), (B2)]</td>
</tr>
<tr>
<td>Class</td>
<td>Method</td>
<td>Private class</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Method</td>
<td>Protected class</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Object</td>
<td>Reference</td>
<td></td>
</tr>
<tr>
<td>Class X Method</td>
<td>Class X Method</td>
<td>Method to method</td>
<td>[(A), (A1), (A2), (A3), (B1), (B2)]</td>
</tr>
</tbody>
</table>

Table 2: Basic Z notations used to formalize the module-level metric

<table>
<thead>
<tr>
<th>Module</th>
<th>Class</th>
<th>Module-class association</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td></td>
<td>(A), (A1), (A2), (A3), (B1), (B2)</td>
</tr>
</tbody>
</table>

Table 1: Generated BBG relation sets

Formalization of module metrics using Z and BBG

The design state space makes it possible to link a metric definition explicitly to the design feature (or features) in BBG. These design features are easy for programmers and analysts to understand and they underline the metric's measurement objective. Linking a metric definition to the design features in the formalized metrics eliminates potential misunderstanding and misinterpretations of the metrics.

We use the Module Interaction Index Metric (MII) metric as an example to show how to formalize module-level metrics. The MII metric calculates how frequently the methods listed in a module's APIs, both Service API (S-APIs)\(^1\) and Extension API (E-APIs)\(^2\), are used by the other modules in the system. MII is defined as follows [11]:

\[
MII(p) = \frac{\bigcup i \in ExtCallRel(i)}{|ExtCallRel(p)|}.
\]

\[MII(S) = \frac{1}{p} \sum MII(p)\]

\(I(p)\) denotes all of the APIs for a module \(p\). For an API \(i \in I(p)\), let \(ExtCallRel(i)\) be the set of calls made to any of the methods of \(i\) from methods outside of the module \(p\).

\[ExtCallRel(i) = \{< m_j, m_i > | Call(m_j, m_i) \in M(i) \land m_j \notin M(p)\}\]

\[ExtCallRel(p) = \{< m_j, m_i > | Call(m_j, m_i) \in M(p) \land m_j \notin M(p)\}\]

In an ideal state, all of the external calls to module \(p\) should take place only through its officially designated API methods. The MII(S) value ranges from 0 to 1. A max MII(S) value of 1 indicates an ideal system where all intermodule interactions are only through the officially designated S-API methods. A min MII(S) value of 0 is indicative of a system with very bad intermodule interaction.

A structural metric is denoted by a characteristic set. The cardinality of the characteristic set equals to the metric value. A characteristic set contains syntactical elements that share certain characteristics. For example, the set of red buses is a characteristic set because all the elements in the set are red buses. In other words, the elements in a characteristic set are not arbitrarily chosen. Table 2 introduces the basic Z notations that are used to formalize the module-level metric we used to validate our formalized framework.

---

1 A Service API (S-API) declares the services that the module provides to the rest of the software system [12][11].

2 An Extension API (E-API) is a declaration of what functionality needs to be provided by an external plugin for the module.
We define the characteristic sets of the MII metric using the Z notations as follows:

**[CLASS, METHOD]**

**CLASS** is the basic set of all classes, **METHOD** is the basic set of all methods in a program, and **MODULE** is the basic set of all modules in a program.

```
Z formalization

We will use the Java example (Listing 1) to illustrate the Z formalization of the MII metric based on the BBG relation sets.

1. First, we get all modules in the design, using the BBG relation set ModuleClass, which is the set of all module-class relations defined in the design. We assign the extracted set to the variable MoC. The MoC set for the example returns: \{(A, A1), (A, A2), (A, A3), (B, B1), (B, B2), (B, B3)\}

2. Then, we collect the API methods, defined as public in the module. Formally defined as $\text{ClassAPIMethod} = (\text{ClassPublicInstanceMethod} \cup \text{ClassPublicClassMethod})$. The two sets involved in the union are basic BBG relations. For our example, $\text{ClassAPIMethod} = \{(A, \text{GetAPublicMethods}), (A, \text{SetAPublicMethods}), (A, \text{MA11}), (A, \text{MA12}), (A, \text{MA13}), (A, \text{MA21}), (A, \text{MA22}), (A, \text{MA23}), (A, \text{MA24}), (A, \text{MA31}), (A, \text{MA32}), (A, \text{MA33}), (A, \text{MA34}), (B, \text{GetbPublicMethods}), (B, \text{SetbPublicMethods}), (B, \text{MB11}), (B, \text{MB12}), (B, \text{MB13}), (B, \text{MB21}), (B, \text{MB22}), (B, \text{MB23}), (B, \text{MB24}), (B, \text{MB31}), (B, \text{MB32}), (B, \text{MB33}), (B, \text{MB34})\}$.

3. Next, we collect the method-to-method messaging in each class. This is represented by the BBG MethodToMethodMessage relation set. In the example, $\text{MethodToMethodMessage} = \{(\text{A, MA11}), (\text{A, MB11}), (\text{A, MA13}), (\text{B, MB21}), (\text{A, MA23}), (\text{B, MB23}), (\text{A, MA32}), (\text{B, MB31}), (\text{B, MB11}), (\text{A, MA12}), (\text{B, MB13}), (\text{A, MA33}), (\text{B, MB22}), (\text{A, MA33}), (\text{B, MB33}), (\text{A, MA21})\}$.

4. To calculate the external calls into the module, we use these steps in the Z schema definition:

   a) “ran({module} \leq \text{MoC})” domain restricts the set of modules in the design by a specific module, then get the range set. For our example, applying this on module A would result in:
   i. \{A\} \leq \text{MoC} = \{(A, A1), (A, A2), (A, A3)\}.
   ii. Then, ran(A \leq \text{MoC}) = \{A1, A2, A3\}.

   The result assigned to variable MC.

   b) Get the methods in all classes, represented by the set $\text{ClassAPIMethod}$ calculated before for our example.
c) Domain restrict ClassAPIMethod by MC – in Z syntax this is formally written as ClassAPIMethod \subseteq MC - to get the class methods pairs for the module(s) that of interest to this example. (ClassAPIMethod \subseteq \{(A1, A2, A3)\}) = \{(A1.GetAPublicMethods), (A1.SetAPublicMethods), (A1,MA11), (A1,MA12), (A1,MA13), (A2,MA21), (A2,MA22), (A2,MA23), (A2,MA24), (A3,MA31), (A3,MA32), (A3,MA33), (A3,MA34)\}. The result set is assigned to variable MIC.

d) Next, we get all of the external calls by the methods in the set MIC. By domain restricting the BBG relation set MethodToMethodMessage by MIC calculated in (c) above. (MethodToMethodMessage \subseteq MIC) = \{((A1, MA11), (B1,MB11)), ((A1,MA1), (B2,MB21)), ((A2,MA23), (B2,MB21)), ((A3,MA32), (B3,MB31))\}. Result is assigned to variable MethodsExtCalls.

e) Then, we get only the range from the set ran(MethodsExtCalls) = \{(B1,MB11), (B2,MB21), (B3,MB31)\} with cardinality value 3.

f) This represents ExtCallRef(p) in the original MII definition, where p is the module A under measurement in the design.

5. To calculate the external calls to a single method in a module, we use these steps in another formal Z schema definition:

a) The input will be the class-method pair, which we want to calculate the external calls for. For example, we want to know the external calls for (A1, MA11).

b) We get all external calls to the class-method pair. By domain restricting MethodToMethodMessage by the class-method pair passed.

MethodToMethodMessage \subseteq \{(A1, MA11)\} = \{((A1, MA11), (B1,MB11))\}. The result is assigned to variable MethodsExtCall (note: this is different from the variable MethodsExtCall used in 4.d before).

c) Then, we get only the range from the set ran(MethodsExtCalls) = \{(B1, MB11)\} with cardinality value 1.

d) This represents ExtCallRef(i), where i is an API method in a module p to represent the external calls to this method only, and not all of the methods in the module. For the previous steps, i represents the pair (A1, MA11).

The same steps 5.a to 5.d will be repeated for the remaining class-method pairs in module A. These are represent in this set \{(A1,GetAPublicMethods), (A1.SetAPublicMethods), (A1,MA11), (A1,MA12), (A1,MA13), (A2,MA21), (A2,MA22), (A2,MA23), (A2,MA24), (A3,MA31), (A3,MA32), (A3,MA33), (A3,MA34)\}. And the result would be \{(B1,MB11), (B2,MB21), (B3,MB31)\} with cardinality 3.

Therefore, the set \{(B1,MB11), (B2,MB21), (B3,MB31)\} represents the characteristics set for the metric Module External Access (MEA), which in our example was applied to module A. The result represents the frequency that methods in a particular module are being accessed by methods in other modules.

And the set \{(B1,MB11), (B2,MB21), (B3,MB31)\} represents the characteristics set for the metric Method External Access (MHEA), when applied to a particular method in a module, to represent the frequency that this method is being accessed by methods in other modules. This completes the Z formalization of the MII metric.

The Design State Space Toolkit

We implemented a new toolkit called the Design State Space (DSS) toolkit to validate that the proposed framework is practical and useful.

The DSS Parser utilizes the JavaCC technology [9] and the SableCC compiler compiler [6] to support the parsing and extraction of the design state space. The DSS Analysyer serves two main purposes. First, it provides the mapping of the parsed tree sets from the source code into the BBG relation sets as the intermediate representation. Second, it extracts definitions, such as structural metrics, from the BBG sets. These relations sets provide convenience and flexibility for all kinds of analyses and future plug-in modules to the tool. For example, we can use these sets to extract structural metrics, if the structural metrics are formalized in the framework. If a new metric definition is provided through the plug-in interface, the tool can extract that metric without changing anything else in the software design.
Figure 1 shows a high-level architecture of the DSS toolkit. For a more detailed discussion of the DSS toolkit and the different modules, please refer to [1].

Conclusions and future research

In this paper, we presented the Z formalization of an object-oriented module-level metric. This formal framework links metrics definitions to the design structures in a graph-based design representation (BBG) that is formalized by the Z specification language. Using the proposed approach, we formalized the module metric: Module Interaction Index (MII). In the future, we plan to formalize more module-level metrics of different complexity to further validate the proposed framework.

References


