Random Application Generator (RAG)

The purpose of RAG is to generate syntactically correct Java programs of arbitrary complexity using the notion of stochastic parse trees (SPT) [1]. The idea of SPT is that a language grammar is represented as an Abstract Syntax Tree (AST) with its nodes assigned different probabilities. The probability for mandatory syntactic grammar features is always one, while optional features have probabilities between zero and one.

Consider the following AST for a part of the Java grammar as it is shown in Figure 1. Class declaration is a mandatory feature of Java since every Java program must have at least one class. However, method declaration is an optional feature, a class may have zero or more methods. Thus, we can assign some probability to generate methods in a class, in theory RAG should determine itself what methods to generate and add them to a given class.

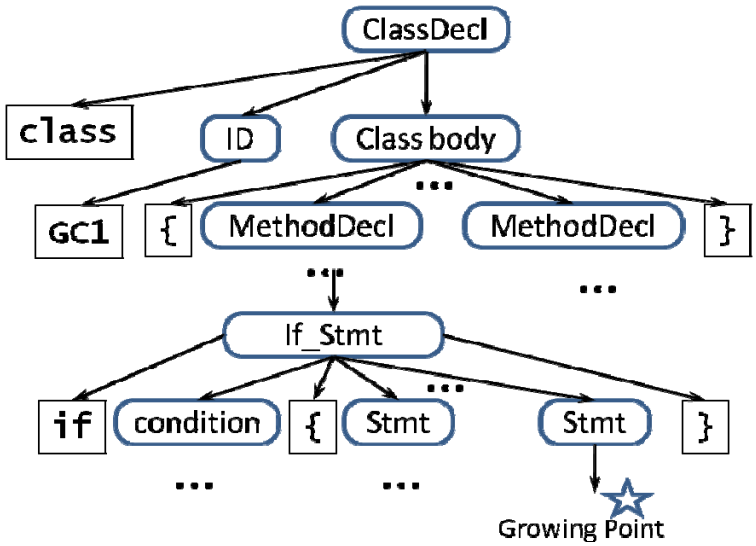

Figure 1. A part of the AST for Java grammar.

RAG will employ a top-down approach for generating Java applications. At the absolute minimum, a Java application must have one class that has the method `main`, which may or may not instantiate the class and invoke a method of this class. Other Java language features are added by RAG based on configuration parameters that are defined by the RAG's users. For example, if the user specifies that the generated application should have between 10 and 100 classes and between 10,000 and 100,000 Lines of Code (LOC), then RAGs will generate a random number C between 10 and 100 and a random number

[1] D. Slutz. Massive stochastic testing of SQL, In *Proceedings of the Twenty-Fourth International Conference on Very-Large Databases*, Morgan Kaufmann, pp. 618–622, 1998.

L between 10,000 and 100,000. Then, RAG will create C classes in the Java applications, adding other features to it until the size of the code, LOC >= L. Because of the randomness, it is really difficult to keep the LOC equal to a specific integer. We could instead, fix the minimum value for LOC (e.g 100) and require that generated lines of code should not be more than 10% of the specified minimum value (e.g. 100 < loc < 110). This is the essence of how RAG works.

The rest of the paper describes Java language features that RAG should add to generated programs and how RAG should do it. All fields are public and static; generics, input/output except for JDBC, and reflection are not used. No concurrency will be used at this point.

Interfaces

RAG takes a configuration parameter that specifies the minimum and maximum numbers of interfaces and the minimum and the maximum number of methods per interface. RAG should have a separate properties file where all the parameters will be specified. The names of interfaces are unique within the generated application. The random generator picks the numbers of interfaces and their methods within the limits of the configuration parameters. When generating a class, RAG will randomly decide what interfaces this class implements and it will add the corresponding interface methods to this class. Each generated interface must be implemented by some class. Abstract classes should be generated and used too.

Inheritance

RAG takes a configuration parameter that specifies the minimum and maximum numbers of classes in inheritance hierarchies, the minimum and maximum numbers of inheritance hierarchies, and the minimum and the maximum number of virtual methods that are overridden when inheriting classes. The names of classes are unique within the generated application. The random generator picks the numbers of classes and their overridden virtual methods within the limits of the configuration parameters.

Suppose that RAG generated ten classes, C1 to C10. Suppose that RAG selects that the application will have one inheritance hierarchy of depth three. As a result, RAG can produce the following hierarchies:

a) class C3 extends C4
b) class C4 extends C8
c) class C8 extends C7

Fields

RAG takes a configuration parameter that specifies the minimum and maximum numbers of fields per class. The names of fields are unique within the generated application. The random generator picks the numbers of fields and their types within the limits of the configuration parameters. A field can be static or nonstatic.

Each field is randomly given its type and it is randomly decided if a field is an array. The type of a field is randomly selected as one of the Java basic types (i.e., char, byte, short, int, long, float, or double), String, or as one of the generated classes or interfaces (since each interface is implemented by some class). RAG should have a configuration parameter to specify allowed types, including a boolean configuration

parameter to set if array type is allowed or not. This way we can restrict the types in generated code to meet Dsc or other symbolic engines' limitation. Each of the generated fields must be used in some expression in the generated Java application.

Methods

RAG takes a configuration parameter that specifies the minimum and the maximum numbers of methods per interface or class, and the minimum and the maximum numbers of parameters per method. RAG randomly decides the return type of the method as well as the types of the input parameters by selecting them as one of the Java basic types (i.e., char, byte, short, int, long, float, or double), String, or as one of the generated classes or interfaces (since each interface is implemented by some class). The return statement of the method will return a local variable or a field whose type matches the declared return type. RAG randomly generates the body of a method by putting statements and expressions in it. Methods can be static and nonstatic.

Method variables

RAG takes a configuration parameter that specifies the minimum and maximum numbers of variables per methods. The names of variables are unique within the scopes of the generated methods. The random generator picks the numbers of variables within the limits of the configuration parameters. RAG randomly decides the type of a variable by selecting it as one of the Java basic types (i.e., char, byte, short, int, long, float, or double), String, or as one of the generated classes or interfaces (since each interface is implemented by some class).

Statements

RAG takes a configuration parameter that specifies the minimum and maximum numbers of statements per methods. These statements can be nested at an arbitrary level. It is desirable that a user specifies the probability with which nesting can happen, thus forcing RAG to add more or fewer nested statements. RAG will generate the following statements that contain different expressions.

a) the if statement with else and elseif branches, the conditions of each of which may contain expressions of arbitrary complexity.
b) the switch statement
c) Loop statements: the for statement, the while statement, and the do-while statements. These loops can contain the break statement, the continue statement, or the throw statement. The latter requires the declaration to the function to match the type of the thrown exception.

Expressions

RAG takes randomly choose what type of expression to pick from below to put this expression in one of the generated statements.

a. array access e[...]
b. nonstatic field access o.f
c. static field access C.f
d. instance method call o.m(...)

e. static method call C.m(...)
f. postincrement e++ and preincrement ++e
g. postdecrement e-- and predecrement --e
h. Negation -e
i. Bitwise complement ~e
j. Logical negation !e if e is of type boolean
k. array creation new e[...]
l. object creation new C(). C can be one of the inherited types.
m. Multiplication e*e, division e/e, remainder e % e.
n. Addition, string concatenation e + e
o. Subtraction e - e
p. Logical expressions: e < e, e <= e, e > e, e >= e, e == e, e != e, e && e, e || e, e instanceof t
q. Assignment x = e

Database Connectivity

RAG takes a configuration parameter that specifies the database that the generated application will use. RAG will generate JDBC code that enables the generated program to connect to the database and execute SQL queries against it. An example of such boilerplate code is shown in Figure 2.

```
public class DBUtil {

    private static final String _userName = "name"; //from a config file
    private static final String _password = "password";//from a config file
    private static final String _driver = "org.apache.derby.jdbc.ClientDriver";//from a config
    private static final String _database ="UnixUsage"; //from a config file

    //return a resultset for data manipulation
    public static ResultSet executeQuery(String sql) throws Exception {
        Class.forName(_driver).newInstance();
        Properties props = new Properties(); // connection properties
        props.put("user", _userName);
        props.put("password", _password);
        Connection conn = DriverManager.getConnection(
                    "jdbc:derby://localhost:1527/"+_database + "", props);
        PreparedStatement ps = conn.prepareStatement(sql);
        return ps.executeQuery();
    }

    //returns a resultset for a fixed number of rows.
    public static ResultSet executeQuery(String sql, int maxRows) throws Exception {
        Class.forName(_driver).newInstance();
        Properties props = new Properties(); // connection properties
        props.put("user", _userName);
        props.put("password", _password);
        Connection conn = DriverManager.getConnection(
                    "jdbc:derby://localhost:1527/"+_database+"", props);
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setMaxRows(maxRows);
        return ps.executeQuery();
    }
```

```
    //adding rows/updating to the database.
    public static boolean execute(String sql) throws Exception {
        Class.forName(_driver).newInstance();
        Properties props = new Properties(); // connection properties
        props.put("user", _userName);
        props.put("password", _password);
        Connection conn = DriverManager.getConnection(
               "jdbc:derby://localhost:1527/"+_database+"", props);
        PreparedStatement ps = conn.prepareStatement(sql);
        return ps.execute();
    }
}
```

Figure 2. Example of the boilerplate code for RAG to add to the generated application.

SQL queries that RAG will execute will insert data into the database, update data, or select data. The latter means that RAG should randomly select the results from a recordset object and assign selected results to variables or fields in the generated application.