TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications

Haruto Tanno
NTT Laboratories, Japan.

Xiaojing Zhang
NTT Laboratories, Japan.

Takashi Hoshino
NTT Laboratories, Japan.

Koushik Sen
EECS Department
UC Berkeley, CA, USA.

***Abstract*—We present CATG, an open-source concolic test generation tool for Java and its integration with TesMa, a model-based testing tool which automatically generates test cases from formal design documents. TesMa takes as input a set of design documents of an application under test. The design documents are provided in the form of database table definitions, process-flow diagrams, and screen definitions. From these design documents, TesMa creates Java programs for the feasible execution scenarios of the application. CATG performs concolic testing on these Java programs to generate suitable databases and test inputs required to test the application under test. A demo video of the tool is available at https://www.youtube.com/watch?v=9lEvPwR7g-Q.**

I. Introduction

Testing is the only predominant technique used by the software industry to make software reliable. Study [2] shows that testing accounts for more than half of the total software development cost in industry. Model-based testing [5] is an emerging area of testing that uses models of software systems for automated test case generation and evaluation. The key idea behind model-based testing is to check whether a system implementation conforms to a model of the system.

We present a model-based testing tool, called TESMA, that we have developed at NTT Laboratories. NTT Group's test design tool, which has been applied to test more than 50 enterprise systems internally, uses the core part of TESMA. TESMA takes as input a design model in the form of screen definitions, business logic definitions, and database table definitions. TESMA transforms an input design model into a directed graph, where the nodes and edges of the graph are annotated with database operations and constraints on the data. From the directed graph, TESMA generates a set of Java programs, where each Java program encodes the behavior of a path in the directed graph.

In this paper, we focus on CATG, an automated test data generation tool that is used by TESMA for test data generation. CATG is an open-source concolic testing [6], [8], [3], [4] tool for Java. CATG has builtin models for database tables and SQL queries. CATG performs concolic testing on the Java programs generated by TESMA. The test inputs and initial databases generated by CATG from the Java programs are transformed into test cases for the application under test. CATG introduces a novel annotation mechanism that enables a programmer to prune the search space of concolic testing.

Note that TESMA generates Java programs instead of performing symbolic execution directly on the directed graph. There are two reasons behind this design decision. First, we wanted to decouple the automated test data generation mechanism from TESMA so that the development of the automated test data generation tool can proceed independently. Second, in order to describe the behaviors of the model, we wanted a formal programming language (e.g. Java) for which automated test generation techniques, such as symbolic execution [7], [1], are well-understood.

The main contributions of this paper are as follows. First, we show the integration of a model-based testing tool and a concolic testing tool to generate test data for enterprise web applications. In our demonstration, we will apply our tools to a toy application and an open-source Java pet store application. Second, we have made CATG publicly available at `https://github.com/ksen007/janala2/` under the open-source BSD license. The CATG distribution contains a pure Java model of a subset of SQL queries and several benchmark programs which can be used to potentially evaluate other Java symbolic execution engines. CATG uses a novel annotation mechanism to guide path exploration in concolic testing.

II. TESMA

TESMA is a model-based test automation tool developed for testing enterprise web applications. TESMA takes as an input a formal design model consisting of three components:

1) screen definition which consists of screen elements (e.g. fields, buttons, links etc.), input constraints (e.g. range of values that can be entered in a field), and events describing the user events,
2) business logic definition which consists of flowcharts (see Figure 2 for an example) describing the behavior of business logic, and
3) database table definitions.

Screen definitions, database table definitions, and business logic details are provided as excel spreadsheets. Flowcharts are described as visio documents. In one of our case studies, for the open-source Java pet store application, we created a design model consisting of 3 screens and 5 database tables. In another case study involving an enterprise system, we had a design model consisting of 31 screens and 42 database tables.

TESMA generates a set of test cases from an input design model. A test case consists of an initial database, screen transitions, input values, and expected results. The test cases are generated as excel spreadsheets. For the Java pet store

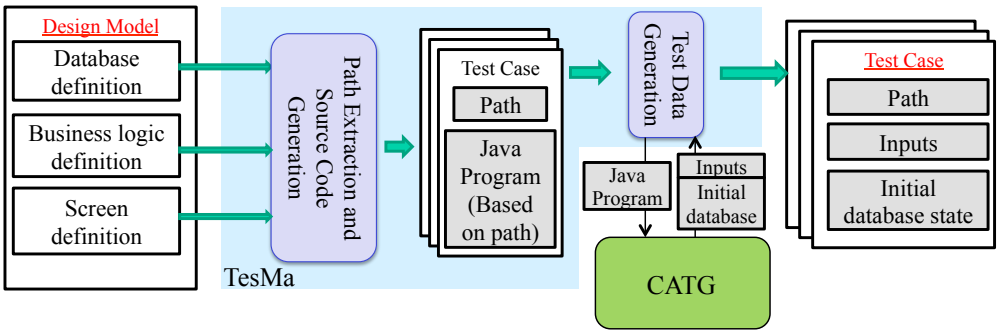

Fig. 1. TESMA workflow

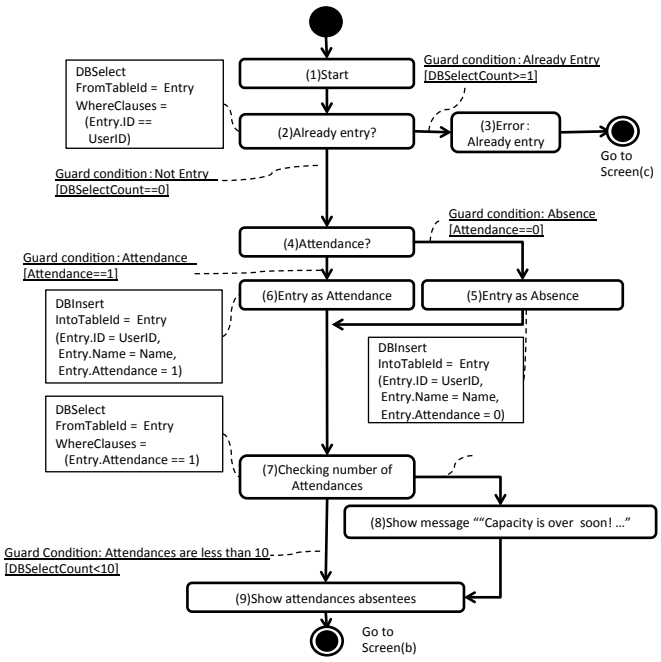

Fig. 2. Business logic definition

```
//Initial records number for each DB table
static int ENTRY_TABLE_SIZE=20;

//Function for checking DB schema constraint
//(Primary key and foreign key)
boolean CheckDBConstraints(...){
  …
}

boolean F(){
  //DB table and input variable declaration
  int UserId = ...;
  …
 ArrayList<Entry> EntryList =
       new ArrayList<Entry>(ENTRY_TABLE_SIZE);
  //Check ranges of input variable
  if(!(UserId >= 0)) return false;
  …

  //Check DB schema constraints
  if(!CheckDBConstraints(EntryList))
    return false;

  //(1)Start

  //(2)Already entry?
  resultList = new ArrayList<Entry>();
  foreach(EntryRecord in EntryList){
    if(EntryRecrod.ID == UserID)
      resultList.add(EntryRecord);
  }
  int dbSelectCount = resultList.count();

  //Guard condition after(2) : Not Entry
  if(!(dbSelectCount == 0))return false;

  //(4)Attendance?
  if(!(Attemdamce==1))return false;
  …

  //(6)Entry as attendance
  newRecord = new Entry(UserId, Name, 1)
  EntryList.add(newRecord);
  //After updating DB table, check DB schema constraints.
  if(!CheckDBConstraints(EntryList))
    return false;

  //(7)Checking number of Attendances
  …

  //Guard condition after(7) :Attendances are over 10
  if(!(dbSelectCount >= 10))
    return false;

  //(8)Show message "Capacity is over soon! …"

  //(9)Show attendances absentees

  //Appropriate test data can be obtained
  return true;
}
```

Path = "(1) (2) (4) (6) (7) (8) (9)"

Fig. 3. Generated Java program

design model TESMA generated 97 test cases, whereas for the enterprise system, TESMA generated 743 test cases.

TESMA uses CATG in the test generation process as follows (see Figure 1). TESMA transforms and combines the above three components of a design model into a directed graph (UML activity diagram). TESMA then extracts all paths from the directed graph. If the graph contains a cycle, TESMA extracts two paths: one path where the cycle is taken once and one path where the cycle is skipped. For each such path, TESMA generates a Java program encoding the behavior of the application along that path. Figure 3 shows a Java program for a path in Figure 2. A successful execution of the program on suitable inputs and database returns true, and returns false otherwise. CATG explores the program and generates a set of test inputs and initial databases for the successful paths of the program. TESMA takes these test inputs and database tables and transforms them into test data.

III. CATG

CATG is the next-generation concolic testing tool for Java programs. Concolic testing performs symbolic execution dynamically, while the program is executed on some concrete input values. Concolic testing maintains a concrete state and a symbolic state: the concrete state maps *all* variables to their concrete values; the symbolic state only maps variables that have non-concrete values. Since concolic execution maintains the entire concrete state of the program along an execution, it needs initial concrete values for its inputs. Concolic testing executes a program starting with some given or random input, gathers symbolic constraints on inputs at conditional statements along the execution, and then uses a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative execution path. This process is repeated systematically or heuristically until all execution paths are explored, a user-defined coverage criteria is met, or the time budget expires.

```
import catg.CATG;
public class Testme {
    private static int foo(int y) {
        return 2*y;
    }
    public static void testme(int x,int y){
        int z = foo(y);
        if(z==x)
            if(x>y+10)
                System.err.println("Error"); // ERROR
    }
    public static void main(String[] args){
        int x = CATG.readInt(2);
        int y = CATG.readInt(1);
        testme(x,y);
    }
}
```

Fig. 4. A Simple Java Program Under Test

A. CATG Design

To apply CATG on Java programs, the user needs to specify the inputs to the program using the following API:

```
public class CATG {
    public static int readString(String x);
    public static int readInt(int x);
    ... // input functions for other datatypes
}
```

In addition, CATG provides an API to make a database table symbolic. We describe this API in the next section.

Figure 4 shows a simple Java program containing two inputs `x` and `y`. `int x = CATG.readInt(1)` in the program indicates that the variable `x` is assigned an input and the value of the input in the first execution defaults to 1. Note that the argument 1 to `CATG.readInt` can be replaced with a call to `Random.nextInt()` to make the first input to the program random.

CATG works in two phases. In the first phase, CATG executes the program under test. At runtime, CATG uses the `java.lang.instrument` API to instrument every Java class file of the program under test, except the CATG runtime classes. The instrumentation of CATG inserts a method call before every bytecode instruction. The inserted methods are implemented by the CATG runtime and they simply log the name and the arguments of the instruction being executed to a trace file. If an instruction loads a value from the memory, CATG also logs the value being loaded in the trace. These concrete values are used by CATG to simplify complex symbolic expressions during concolic execution. At the end of an execution of the program under test, CATG generates a trace file containing all the instructions being executed. A snippet of the trace file generated by CATG on the program in Figure 4 is shown is Figure 5.

In the second phase, CATG re-executes the instructions in the trace file symbolically and generates a path constraint. During the symbolic execution, CATG maintains a symbolic heap and a symbolic call stack. CATG only executes the instructions that are present in the trace. Therefore, if CATG skips the instrumentation of certain class files (such as library classes) in the first phase, the instructions from those classes won't be present in the trace and CATG will not symbolically

```
ICONST_2
INVOKESTATIC owner=catg/CATG name=readInt desc=(I)I
ILOAD var=0
GETVALUE_int v=2
INVOKESTATIC owner=janala/Main name=readInt desc=(I)I
INVOKEMETHOD_END
GETVALUE_int v=2
ISTORE var=1
ILOAD var=1
GETVALUE_int v=2
...
...
```

Fig. 5. A Simple Java Program Under Test

execute those instructions. This could result in stale (incorrect) values in the symbolic state. CATG uses the values logged in the trace to get rid of stale values.

A key advantage of running CATG in two phases is that the symbolic execution of a path is done separately from the concrete execution of the same path. This ensures that the concrete execution of the program is not affected by any bug or side-effect from the symbolic execution. The trace file generated in the two phase process also helps in deterministic debugging of symbolic execution—one could repeatedly run symbolic execution on the trace with the same outcome. The intermediate trace file also gives us the flexibility to implement symbolic execution in any language, though we currently implement symbolic execution in Java. Note that in existing dynamic symbolic execution systems, both concrete and symbolic executions are performed side-by-side in the same execution and they closely interact with each other.

B. SQL Query Support

The Java programs generated by TESMA from design models extensively use SQL queries. CATG provides a library to model a large subset of SQL queries. The library includes support for SELECT, UPDATE, DELETE, INSERT, GROUPBY, HAVING, AGGREGATE, sub-query, views, composite key constraints, foreign-key constraints. For example, one can create a database table `Customers` with four columns and ten symbolic rows using the library as follows.

```
Table customers = TableFactory.create("Customers",
  new String[]{"Id","Name","PasswordHash","Age"},
  new int[]{Table.INT,Table.STRING,Table.INT,Table.INT},
  new int[]{Table.PRIMARY,Table.NONE,Table.NONE,Table.NONE},
  new ForeignKey[]{null,null,null,null});
SymbolicTable.insertSymbolicRows(customers, 10);
```

Similarly, the following SQL query

```
ResultSet rs = statement.executeQuery(
"select Books.Id, Books.Price " +
"from Books inner join Publishers "+
"on Books.PublisherId = Publishers.Id " +
"where Books.Title= '" + title + "' "+
"and Publishers.Name='"+name+"'");
```

gets transformed into the following Java code which calls methods defined in the library.

```
ResultSet rs = Books.select(new Where() {
  public boolean where(Row[] rows) {
    if (!rows[0].get("PublisherId").equals(rows[1].get("Id")))
      return false;
    if (!title.equals(rows[0].get("Title")))
```

```
      return false;
    if (!name.equals(rows[1].get("Name")))
      return false;
    return true;
  }
}, new String[][] { { "Id", "Price" }, null },
   new Table[] { Publishers }).getResultSet();
```

Once the SQL queries are transformed into Java code, CATG can perform concolic execution of these queries like any other Java code.

C. CATG Annotations for Guided Path Exploration

A key limitation of concolic testing is that the number of feasible execution paths of a Java program grows exponentially with the length of an execution and could be infinite for large programs. Therefore, in practice it is not feasible to explore all feasible paths of real-world Java programs. Numerous heuristics and reduction techniques have been proposed to guide path exploration in symbolic execution and concolic testing. However, these techniques do not work effectively for all kinds of programs. We believe that the programmer of a Java program usually has a good understanding of the program and could help concolic testing to guide path exploration effectively so that concolic testing can achieve code coverage quickly.

CATG provides a set of methods that a programmer can add to the program under test. CATG uses these annotation functions to guide concolic testing only along the paths that are of interest to the programmer and prunes out the other paths. One such simple annotation function provided by CATG is `CATG.assertIfPossible(int pathId, boolean predicate)`. To use this annotation function, the programmer first identifies a set of paths in the program that she is interested in exploring. She then associates an arbitrary unique path id to each such path. For each unique path id, the programmer inserts suitable calls to the function `CATG.assertIfPossible` throughout the program. When CATG is invoked with a given path id, it tries to take a path such that any call to `assertIfPossible` along the path satisfies the following requirement: if the first argument to the call matches the path id, then the second argument must evaluate to true. Any path that violates this requirement is discarded.

CATG provides another set of annotation functions that enables programmers to abstract a block of code. CATG provides the following annotation functions.

```
void CATG.BeginScope()
void CATG.EndScope()
int CATG.abstractInt(int x)
boolean CATG.abstractBool()
... // abstract functions for other datatypes
```

Using these annotation function, we can specify that a code block, say `x = foo();`, is abstract as follows.

```
CATG.BeginScope();
x = foo();
CATG.EndScope();
x = CATG.abstractInt(x);
```

Here we surrounded the code block with `CATG.BeginScope()` and `CATG.EndScope()`. Any variable that is written (or computed) by the code block is then abstracted by using one of the `CATG.abstract*()` functions. An abstract block can be nested within other abstract blocks.

With the above annotations, concolic testing works as follows. We explore the program under test using concolic testing, but avoid exploring any block of code enclosed within `CATG.BeginScope()` and `CATG.EndScope()` (which we will call an abstract block). Any such path that ignores abstract blocks is called an *abstract path*. In an abstract path any call of the form `x = CATG.abstractInt(x)` immediately after an abstract block makes the variable `x` symbolic. Given a path π in the abstraction, the path may not be feasible in the actual program. To check the feasibility of an abstract path, CATG performs path refinement, that is, expands the ignored abstract blocks along the path to get a concretely realizable path whose projection on the program under test is π. Path refinement performs a backtracking search over the path π, finds concrete paths through each abstract code blocks that can be stitched together. Since paths inside nested abstract code blocks are expanded on demand to get a concretely realizable path, we only explore relevant parts of the program path space. This significantly prunes our search while retaining the relative soundness and completeness of concolic testing.

A detailed discussion of these annotations is beyond the scope of this paper. However, we will show some of these annotations in the formal demonstration.

D. Capabilities and Benchmarks

CATG supports linear integer constraints, theory of arrays, and string constraints involving concatenation, length, and regular expression matching. CATG is available at `https://github.com/ksen007/janala2/` under the open-source BSD license. CATG uses the ASM `http://asm.ow2.org/` library for bytecode instrumentation and CVC4 `http://cvc4.cs.nyu.edu/web/` for constraint solving. A number of benchmark programs, including several data-structures, JLex, and NanoXML, with suitable harnesses are also made available with CATG.

References

[1] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS'07*, 2007.

[2] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[3] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice – preliminary assessment. In *ICSE Impact'11*, May 2011.

[4] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[5] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'05*, June 2005.

[7] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03*, Apr. 2003.

[8] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*, Sep 2005.