

Automated Testing for Java-Based Programs with Java PathFinder and T2 Framework

Aziz Yudi Prasetyo¹, Ricky Suryadharna², Ade Azurat³, Belawati H Widjaja⁴
Faculty of Computer Science, University of Indonesia

¹ayp41@ui.edu, ²risu51@ui.edu, ³ade@cs.ui.ac.id, ⁴bela@cs.ui.ac.id

Abstract—this paper presents discussion on the use of automated testing tool for Java-based programs, namely Java PathFinder (JPF) and T2 Framework (T2). Here we discuss the use of both for different purpose, with two different programs were built for examination to one specific tool. Technical approach to testing is also reviewed for each to measure characteristics of a respective tool. As each has distinctive characteristics upon conducting tests, it is proposed that the use of both JPF and T2 are not as substitution to each other; it is rather to be as complement on their use as verification tools.

Index Terms—automated testing, java, verification, model checker.

I. INTRODUCTION

The use of automated testing tool has been in immense need upon software engineering due to its nature of delivering automated session of tests. Despite the fact that in many software engineering practices testing is done by manual testing, the use of automated testing tools for assisting testing phase has become common practice within the past years.

Tests are carried through automated generation of test cases, in relation to tracing of logics lying within programs on which testing is conducted. This approach is done in order to expose any fault or bug within questioned programs. Although not necessarily, automated testing tools may significantly differ on generating test cases as each of them may do such in a distinct fashion. Some of which may adhere to manually given preconditions, as well as the test cases may be completely random-generated.

While an automated testing tool works to verify given program against the specifications it has, it commonly also have the ability to detect and exploit such unspecified in the specifications e.g. runtime exception, I/O error and non-termination cases. In addition, it also checks and reports traces of events leading to a violation of specifications.

This paper discusses the use of two automated testing tools, namely Java PathFinder (JPF) and T2 Framework (T2). Those tools are testing tools developed for Java-based software. Here we explore the characteristics of

both through experiments of verification upon Java-based programs. We will also review basic characteristics and approaches used by each of the respective tools, followed by some experiment report. We discuss behaviors appearing within the testing phase, including common and differing characteristics between both as automated testing tools.

II. JAVA PATHFINDER

Java PathFinder [1] is an automated testing tool developed to verify Java bytecode programs. It systematically traverses possible execution paths within a Java-based program to expose violations of properties such as deadlocks and unhandled exceptions. Additionally, it also reports detailed trace of executions that leads to defects or violations.

Developed based on model-checking tool [2], the use of JPF is mainly aimed for all sort of runtime-based verification purpose. On this perspective, JPF differs from any normal debugger running on a Java Virtual Machine. JPF executes and stores the program states it meets during verification process. Technically, this approach needs a thorough consideration upon the possibility of state explosion problem. This, however, has been tackled by JPF through some heuristics and state abstraction techniques. More on this approach is to be discussed on a later part.

JPF was built as a Java-based application that can be used either as standalone application through command line or inside development environments such as in IDE. As of its nature as a model checker, JPF checks execution paths available of a given program, including cases involving multithreading environment.

A. Basic Properties

JPF is a based on model-checking tool. JPF explores all potential paths traversable defined by program statements. It implies the needs of proper search strategies and efficient memory management – those are configurable as part of JPF.

By defining a particular program as a tree, what JPF does upon verification is considered an analog to a depth-

first search process. Here, backtracking mechanism is allowed to see if there are paths that have yet to be explored, thus render JPF theoretically able to trace almost any possible path within any particular Java-based program when given enough time and memory usage.

One notable feature of JPF is its ability to explore scheduling sequence for the purpose of verification involving multithreading environment. It has to be noted that this area has been particularly difficult for any traditional testing approach. JPF open the possibility for a better program verification in multithreading environment.

B. Approach to Testing

JPF employs a systematic search for all paths traversable within a particular program – which starts from the `main` method as the entry point for any Java-based program. It then employs a depth-first search with its given search strategies and heuristics. Technically, this implies the use of extensive heap space of memory resources, depending on the complexity of a particular program. Obviously, this also implies that verification may take longer (or shorter) time, depending on choices and path possibilities of a program execution.

To anticipate the possibility of state explosion, JPF provides configurable search strategies and reducing states (reducing state storage). This is reached by the use of heuristics upon search strategies and implementation of advanced state abstraction, which reduce the state space significantly. Such implementations of state abstraction include pattern matching based on shape analysis of data structures.

JPF does not necessarily check a program against its own specifications. Rather, it only checks whether there are any violation (i.e. runtime error, exception) in any possible path. Inserting a specification is possible by using assertion within codes. Any violation to corresponding specification will be reported as `AssertionError`. This, however, is not considered as a clean method of inserting a specification to be tested against a built program.

For example, we might want to check a simple method that does multiplication of two positive integers using loop of summations:

```
public int multiply (int a, int b) {
    assert (a >0 && b >0) : "prec";

    int base = a;
    int mult = b;
    for (int i = 1; i<mult; i++)
        { a = base+a; }

    assert a = base*mult : "postc";

    return a;
}
```

Here "prec" and "postc" represents the specification of preconditions and post conditions of the corresponding method. In JPF, specifications have to be

inserted right inside the program code. Although, inserting specifications with such method is effective, yet it may be hassle to do upon verification of predefined specification against built programs.

III. T2 FRAMEWORK

T2 Framework is an automated testing tool developed to verify Java programs in its form as bytecode. Its functionality involves verifying a particular program against a given specifications, including class invariants and method specifications using preconditions and post conditions. It also has to be noted that specifications in T2 can be defined as some formal specifications. Both the program and its specifications are written in plain Java.

T2 is developed as a trace-based testing tool [3]. It verifies a particular program through tracing sequences of randomly generated method calls. Albeit its nature of random generation, this is considered powerful since in sequence of calls, methods are essentially checking each other. More on this behavior is to be discussed on the later part.

Verification using T2 is done by executing the tests through command line, with some modifiable configurations are available as command line arguments. In addition, T2 also reports traces of execution that leads into defects or violation against specifications of the program being verified.

A. Basic Properties

T2 is an automated testing tool that works on verification of a particular program by tracing sequences of method calls. T2 checks whether a corresponding program has defects (i.e. runtime error, unhandled exception) through the trace of the method calls. In advanced testing phases, checking of program specifications can also be implemented by defining the specifications to work with T2.

Specifications in T2 are mainly involving class invariants and method specifications, although some others (such as application model) are also available. These are written and compiled in Java along with the program code – this method is also called as in-code specifications. By using in-code specifications, one can assure that maintenance of both specifications and code are done in a synchronized fashion. Additionally, T2 also provides a convenient method on writing specifications in a wrapper method – thus eliminates the hassle of running through the program code.

It however needs to be noted that T2 has its limitations concerning concurrency and multithreading environment [4]. As of present time, concurrency is beyond the current ability of T2, thus rendering it incompatible with tests of Java-based programs involving concurrent threads. It is unknown whether later version of T2 has yet to support concurrency checks, but it is certain that concurrency is

outside the capabilities of T2 as it is discussed in this paper.

B. Approach to Testing

T2 wraps the specifications of a program subject to verification along with the program code, in which the specifications being compiled along with the corresponding class. The specifications are including class invariants and method specifications, while specifications such as application models and temporal property checks are also supported.

The nature of specification in T2 is as follows: class invariants are defined in a method named `classinv()` with a return type of `boolean`. This is where the class invariants are defined, written in Java. The class invariant is checked for each instance of a new object by T2, and is checked again for each method calls generated in a test session. The method specifications is defined in a wrapper method named `m_spec()` with same return type to the corresponding method `m`.

Suppose that we have a class `C` with method `m` returning an integer. We write the definitions for class invariants and method specifications in T2 as follows:

```
private boolean classinv() {
    return (/*class invariants*/);
}

public int m_spec () {
    assert (/*preconditions*/) : "PRE";
    int ret = m();
    assert (/*postconditions*/) : "POST";
    return ret;
}
```

By using proper specifications, T2 can also check properties of Hoare triple and class invariant in a given program. It is also to be noted that specifications in T2 are unambiguous, declarative (specifying 'what result' instead of 'how to get result') formal (written in Java), and powerful (with exploitations of Java library).

On the other hand, it should also be noted that the specifications are written with less hassle: there is no need to run through program statements to write the specifications to a corresponding method. Additionally, specifications are maintained with minimum cost due to the in-code specifications.

It has been mentioned that T2 verifies a particular program through tracing of randomly generated sequences of method calls. Although it may seem otherwise, this is considered powerful as in such sequences methods are checking each other for its specifications.

For instance, let a sequence in the program code of class `C1` in method `m4()` calls methods such like `m1()`; `m2()`; `m3()`; . In this sequence of codes, not only T2 checks the specifications of `m4()`, but also `m1()`, `m2()`, and `m3()` as well. This asserts that in trace-based testing by T2, we essentially check `C1` against its own

specifications in conjunction with all specifications defined for its methods. In addition, the native fault detection mechanism in Java as well as fault detection mechanism programmed internally inside the class also adds up for the checks done by T2. Combining these altogether implies stronger correctness check, even in cases that each individual specifications are weak.

IV. VERIFICATION ON JPF AND T2

Here we employ two different cases of program verification to T2 and JPF. Both are small programs developed for purpose of exploring the capabilities of JPF and T2 as automated testing tools. We defined a small program as a Java-based program consists of one class, not more than 150 lines of codes.

We divide this section into two main parts; one describes an instance of verification process using JPF for a particular program, while another describes an instance of verification process using T2 for a different program. For each part, discussions are carried through description of program to be verified and the verification process using a respective tool. In addition, a brief analysis is also supplied for each of described verification process.

A. JPF – char-based short summation

We developed a simple program to do a summation of two number of `short` datatype. This however is not done in a normal fashion through numerical addition, but by using operations involving `char` datatype.

The main function of this program lies within the method `add`, which is passed with two numbers of `short` datatype as parameters. This method returns a number of `int` datatype as the summation result of its parameters. The source code follows for the corresponding method.

```
static int add(short a, short b) {
    if(a == 0) {
        return b;
    }
    else if(b == 0) {
        return a;
    }

    String sa = Short.toString(a);
    String sb = Short.toString(b);
    String result = "";

    int ia = sa.length() - 1;
    int ib = sb.length() - 1;

    while(ia < ib) {
        sa = "0" + sa;
        ia++;
    }

    while(ib < ia) {
        sb = "0" + sb;
        ib++;
    }

    // at this time, ia is equal to ib
```

```

assert(ia == ib);

int save = 0, temp;

while(ia >= 0) {
    temp = sa.codePointAt(ia) - '0'
        + sb.codePointAt(ia) - '0'
        + save;
    if(temp >= 10) {
        save = 1;
        temp -= 10;
    }
    else {
        save = 0;
    }

    result += (char)(temp + '0');
    ia--;
}

sa = result + (char)(save + '0');
result = "";

for(ia=sa.length()-1; ia>=0; ia--){
    result += sa.charAt(ia);
}

return Integer.parseInt(result);
}

```

The logic in the program follows that the two numbers of short datatype is converted into two variables of String datatype. It then does the summation analog to primitive addition technique by adding column per column while maintaining the carry variable. It however needs to be noted that the program is doing such in a manner of handling char datatype for each digit of number, before finally it returns a variable of String datatype representing a corresponding number. The result is then parsed into a number with int datatype before returned as a summation result between both parameters.

On the other hand, this program also has some sort of specifications coded around the entry point of the program (i.e. the main method) using assertion technique provided by Java.

```

import gov.nasa.jpf.jvm.Verify;

...

public static void main(String[] args) {
    short a = Verify.random(10);
    short b = (short)Verify.random
        (Short.MAX_VALUE);

    assert((a + b) == add(a, b));
}

```

For this purpose, JPF provided a specific random number generator that is defined in package gov.nasa.jpf.jvm.Verify. In this case, the package was solely used for generating random numbers required for verifying the program in question.

It is obvious that without the specification defined by

the assertion, JPF would not be able to expose the faults within the program logic. While it might be possible to expose runtime errors such as NullPointerException or NumberFormatException, the main part of the verification upon this program lies within the specification that $a + b$ is equal to $\text{add}(a, b)$.

The program was tested using JPF, and has proved to be correct. However, the verification process took an excessive amount of time given the result for the simple program. This is due to use of the method random in gov.nasa.jpf.jvm.Verify in conjunction with the +vm.enumerate_random=true as the argument for JPF. This argument ensures that JPF will enumerate all possibilities upon random number generation, thus rendering the verification process taking excessive amount of time. However, this option is considered as more secure, given that all possible paths and all possible combination of numbers are explored through verification by JPF.

B. T2 – Newton method for quadratic equation

A simple solver for quadratic equation was introduced for the purpose of verification. Here we have a program whose purpose is to print the roots of a quadratic equation $ax^2 + b + c = 0$, given arbitrary coefficient of a, b, and c.

The program consists of class QuadraticSolver, whose constructor described as follows. Some declarations and supporting methods are not included for the sake of simplicity.

```

public QuadraticSolver(String sa, String
sb, String sc) {

    a = Double.parseDouble(sa);
    b = Double.parseDouble(sb);
    c = Double.parseDouble(sc);

    roots = new double [2];
    diff = new double [2];

    System.out.println("a = "+a
        +" b = "+b+" c = "+c);

    switch (discriminant(a,b,c)) {

    case ONE_ROOT:
        roots[0] = rootOf(a,b,c);
        roots[1] = roots[0];

        break;

    case TWO_ROOTS:
        roots[0] = rootOf(a,b,c);
        diff[0] = adiff;

        do {
            roots[1] = rootOf(a,b,c);
            diff[1] = adiff;

        } while (Math.abs(roots[0]-
            roots[1])< D_VALUE);
        break;
}

```

```

case NO_REAL_ROOT:
    System.out.println("no real root");
    return;
}

System.out.println("roots are: "
    +roots[0]+" and "+roots[1]);
System.out.println("diff#0 = "
    +diff[0]);
System.out.println("diff#1 = "
    +diff[1]);
}

```

The main function of the program lies within the method `rootOf` that accepts three parameters of double datatype, representing coefficient of a , b , and c respectively. The method is described as follows:

```

public double rootOf(double a, double b,
double c) {

    x = new java.util.Random().nextInt();
    c = c/a; b=b/a; a = 1;

    do {
        d = (Math.pow(x, 2)+b*x+c)/(2*x+b);
        x = x-d;
    } while (Math.abs(d) >= D_VALUE);

    return x;
}

```

In the method `rootOf`, we have the Newton method represented in Java. Here we have the loop guard defined that the loop will not terminate until x_{n+1} and x_n are having values that differ below some threshold. This threshold is defined as a constant integer, referred as **D_VALUE** within the corresponding code (see the highlighted parts of code above).

On to the specification, we have specified a preconditions and post conditions only to the method `rootOf`. On a side note, we also have supporting method for determining a discriminant value for quadratic equations, but such method is considered as rather simple as it contains only basic operations of addition and multiplication.

Here we have the specification of the method `rootOf`. The precondition and post condition are marked with "PRE" and "POST" respectively, in accordance that T2 puts into accounts of what kind of specifications to be checked by its internal engine.

```

public double rootOf_spec(double a,
double b, double c) {
    assert a != 0 : "PRE";
    double d = rootOf(a, b, c);
    assert Math.abs(a*(Math.pow(d, 2)) +
        b*d + c) < E_VALUE : "POST";
    return d;
}

```

In the method specification `rootOf_spec`, we also have a constant defining the accuracy needed as post condition. Here it is denoted as **E_VALUE**, in which it holds the condition whether the program is valid on its verification by T2.

Now we employ verification using T2 for assessing the behavior of the program against different combinations of **D_VALUE** and **E_VALUE**. For this purpose, we divide both **D_VALUE** and **E_VALUE** into classification of high accuracy (**h_acc**) and low accuracy (**l_acc**).

Constant	h_acc	l_acc
D_VALUE	10^{-10}	10^{-2}
E_VALUE	10^{-10}	10^{-4}

For each combination of **D_VALUE** and **E_VALUE**, a verification process is carried through T2. We then assess the behavior of the questioned program through the result yield by T2. The results are as follows.

#	D VALUE	E VALUE	Result w/ T2
c1	l_acc	l_acc	valid
c2	l_acc	h_acc	invalid
c3	h_acc	l_acc	valid
c4	h_acc	h_acc	valid

The results yield the program valid for all combinations, except for #c2. This is due to the nature of **E_VALUE** being as high accuracy is combined with the **D_VALUE** being set as low accuracy. Given that **E_VALUE** is part of the post condition of method `rootOf`, the program must yield a high accuracy result to render itself valid. This, however, is not applicable given that the **D_VALUE** is at low accuracy nature. As the **D_VALUE** holds the loop guard for the Newton method, attaining result of high accuracy is more unlikely to happen. In contrast, **E_VALUE** at high accuracy nature implies post condition for method `rootOf` to be strictly accurate, and this render the verification for #c2 invalid.

V. DISCUSSION AND ANALYSIS ON JPF AND T2

We have assessed the behavior of JPF and T2 as automated testing tools, by carrying verification upon Java-based programs using both. The results of experiments should be adequate to yield general perspective on capabilities of either JPF or T2 as means upon carrying program verification.

Upon the verification process carried in the previous part, JPF has shown to be powerful as an automated testing tool. It independently traverses its way in all possible paths of a particular program being verified, thus making it versatile as a model-checking tool. However, it faces the issues of heap memory management as well as the excessive amount of time it may take during verifications of advanced or more complex program.

On the other side, T2 offers a more formal approach while it still adheres with the syntactic definitions in Java. As in the verification process described earlier, T2 provides a clean and convenient way to write and test specifications to a particular program. The support of formal specifications was also found to be suitable for verifications involving mathematical purpose or logical inference. However, T2 suffers a significant drawback that it has yet to support verifications on multithreading environment when it is to be compared with JPF.

Still, it is to be emphasized that the verification process carried and reviewed here is not by any means an adequate reference to represent the full capabilities of either JPF or T2. A more thorough series of experiments involving comparison and technical assessment about both would be needed for such purpose to yield a non-biased and verifiable result.

VI. CONCLUSION

JPF and T2 are of different type in terms of automated testing tools, thus comparison between both needs to be carefully assessed and analyzed. This paper is by no means intended to deliver a thorough comparison and analysis in terms of advantages and drawbacks between both, thus it is not to be viewed in such ways of understanding.

JPF, on one hand was developed as a model checker, and it takes approach upon program verifications by developing a model and examining all possible paths and outcomes of a given program. While its approach is less formal and seems to be more technical in use, JPF has its own basis of formal methods, given that the nature of a model-checking tool relies very much upon formal methods.

On the other hand, T2 has a more formal approach on its use. It supports definition of program specifications in a more formal fashion, along with a convenient method to write and maintain them. It should also be noted though, that although it may not be fully black box (as it still needs to access corresponding source code of verified program), T2 has most of its characteristics leaning towards a black box testing tool.

Considering the characteristics between both, using JPF and T2 in conjunction might possibly lead to overheads of additional cost and resource on a testing phase of software development. However, it should also be noted that despite the nature and characteristics they might share, the use of both are of different areas, thus it is considered to be more proper to view one as complement of another instead of as complete substitution.

REFERENCES

- [1] W. Visser, K. Havelund, G. Brat, S. J. Park, F. Lerda. "Model Checking Programs" Automated Software Engineering Journal, 2nd ed vol. 10, 2003.
- [2] F. Lerda and W. Visser. "Addressing Dynamic Issues of Program Model Checking". Proceedings of SPIN2001. Toronto, May 2001
- [3] I. S. W. B. Prasetya, T. E. J. Vos, A. Baars. "Trace-based Reflexive Testing of OO Programs". Department of Information and Computing Sciences, Utrecht University. 2007.
- [4] I. S. W. B. Prasetya. "T2: Automated Testing Tool for Java, User Manual". Department of Information and Computing Sciences, Utrecht University. 2007.