Effectiveness of Automated Function Testing with Petri Nets: A Series of Controlled Experiments

Dianxiang Xu
Department of Computer Science
Boise State University
Boise, ID 83725, USA
dianxiangxu@boisestate.edu

Ning Shen
Department of Computer Science
Boise State University
Boise, ID 83725, USA
ningshen@u.boisestate.edu

Abstract—Existing work has developed techniques for automated generation of function tests from high-level Petri nets. Yet there is no empirical evidence that demonstrates the cost-effectiveness of this approach. This paper presents a series of controlled experiments to evaluate the fault detection capabilities of various strategies for test generation from high-level Petri nets. We built test models and generated executable test code for three subject programs. Then we executed the test code against more than 300 mutants of the subject programs. Each mutant is a variation of the original subject program with one injected fault. The experiment results show that (a) the reachability coverage-based tests are more effective than the tests for state coverage and for transition coverage, (b) postcondition-based test oracles are more effective than state-based oracles, (c) robustness tests with invalid inputs are critical to improving fault detection capability. In particular, the reachability coverage-based tests together with robustness tests killed 99.7% of the mutants.

Keywords—Software testing, model-based testing, Petri nets, fault injection, test coverage, test oracle

I. INTRODUCTION

Model-based testing makes use of explicit models of a system under test (SUT) for generating test cases and verifying conformance between the SUT and the models [16]. It is an appealing method for function testing of software because of several benefits [14]. First, the modeling activity helps clarify test requirements, which are critical to effective testing. Second, generation of test cases can be automated or partially automated. Automation enables more test cycles and assures the required coverage of test models. Third, model-based testing can help improve fault detection capability due to the increased number and diversity of test cases.

Among the various modeling formalisms for model-based testing, UML state machines are the most popular. Empirical studies have shown that testing with state machines can detect many faults [1][14]. Nevertheless, the expressiveness of UML state machines is limited because all states need to be enumerated explicitly and it is difficult to specify and reason about test data. To address these issues, we have demonstrated that high-level Petri nets can be an expressive formalism for model-based testing because they can represent both control flows and data flows. Specifically, we have developed a tool, MISTA (Model-based Integration and System Test Automation), for generating executable tests from function nets, which are lightweight high-level Petri nets [12][13]. It can generate tests to meet a given coverage criterion, such as state coverage, transition coverage, and reachability coverage. It also provides an expressive way for describing the relations between the model-level elements and the implementation-level constructs in the target language or test environment so as to automatically transform model-level tests into executable code. MISTA supports not only various programming and scripting languages (Java, C#, C, C++, HTML, Python, and VB), but also a number of test execution frameworks, such as xUnit for testing Java and C# programs, Robotium for testing Android mobile applications, Selenium IDE for testing web applications, and Robot Framework for keyword-based testing.

Our prior work has also applied MISTA to function testing of real-world systems [12]. However, cost-effectiveness of the automated testing remains an open issue. This paper presents a series of controlled experiments to evaluate the cost-effectiveness of various strategies for test generation from function nets. These strategies involve different types of test inputs (normal tests and robustness tests), different coverage criteria of test models (state coverage, transition coverage, and reachability coverage), and different methods for defining test oracles (states as oracles and postconditions as oracles). To the best of our knowledge, this is the first empirical study that evaluates the cost-effectiveness of automated model-based testing with high-level Petri nets.

The remainder of this paper is organized as follows. Section II gives an introduction to function nets and test generation. Section III describes the research questions and experiment setup. Section IV presents the results; Section V reviews related work. Section VI concludes this paper.

II. FUNCTION NETS AND TEST GENERATION

A. Function Nets

Function nets in MISTA are a lightweight version of high-level Petri nets: predicate/transition nets [3] and colored Petri...
nets [4]. It improves usability yet maintains the expressiveness of high-level Petri nets. A function net \( N \) is a tuple \(<P, T, F, L, \varnothing, M_0, >\), where:

1. \( P \) is a set of places (i.e., predicates), \( T \) is a set of transitions, \( F \) is a set of normal arcs, and \( L \) is a set of inhibitor arcs.
2. \( L \) is a labeling function on arcs \( F \). \( L(f) \) is a label for arc \( f \). Each label is a tuple of constants and variables.
3. \( \varnothing \) is a guard function on \( T \). \( \varnothing(t) \) is a guard condition for transition \( t \).
4. \( M_0 \) is a set of one or more initial markings.

In this paper, multiple initial markings (i.e., states) can be associated with the same net structure. Suppose \( M_0^k \) is an initial marking and \( M_f^k(p) \) is the set of tokens residing in place \( p \). A token in \( p \) is a tuple of constants \(<X>_p \), denoted as \( p(X)_p \). The zero-argument tuple is denoted as \(<\varnothing>_p \). For token \( t \) in \( p \), we simply denote it as \( p \). We also associate a transition with a list of variables as formal parameters, if any.

Let \( p /t \) be a place and transition, respectively. \( p \) is called an input (or output) place of \( t \) if there is a normal arc from \( p \) to \( t \) (or from \( t \) to \( p \)). \( p \) is called an inhibitor place if there is an inhibitor arc between \( p \) and \( t \). Let \( \chi /\chi \) be a variable binding, meaning that variable \( \chi \) is bound to value \( \chi \). A variable substitution is a set of variable bindings. For example, \(<x/l, y/2>\) is a substitution where \( x \) and \( y \) are bound to \( l \) and \( 2 \), respectively. Let \( \theta \) be a variable substitution and \( l \) be an arc label. \( l/\theta \) denotes the tuple (or token) obtained by substituting each variable in \( l \) for its bound value in \( \theta \). For instance, \( l/\theta = <l, 2> \) if \( l = <x, y> \) and \( \theta = <x/l, y/2> \).

Transition \( t \) is said to be enabled by \( \varnothing \) under a marking if:

1. each inhibitor place \( p \) of \( t \) has no token that matches \( l/\theta \), where \( l \) is the label of the inhibitor arc between \( p \) and \( t \); and
2. each inhibitor place \( p \) of \( t \) has no token that matches \( l/\theta \), where \( l \) is the label of the inhibitor arc between \( p \) and \( t \); and
3. the guard condition of \( t \) evaluates true according to \( \theta \).

Suppose an initial marking for the function net in Figure 1 is \( \{holding(1), clear(2)\} \). \( \text{stack} \) is enabled by \( \theta = <x/l, y/2> \) because token \( <1> \) in the input place \( holding \) matches \( <x/l, \theta > \), token \( <2> \) in the input place \( clear \) matches \( <y/\theta > \), there is no token in the inhibitor place \( handempty \) that matches \( <e>_x \), and the guard condition \( x/\chi = y \) is true according to \( \theta \).

Enabled transitions can be fired. Firing \( t \) with \( \theta \) under \( M_0^k \) leads to a new marking \( M_0^k(t) \) as follows:

1. For each input place \( p \) of \( t \) with input arc label \( l \), the unified token \( l/\theta \) is removed from \( p \), i.e., \( M_f^k(p) = M_f^k(p) - \{\theta(l/\theta)\} \).
2. For each output place \( p \) of \( t \) with output arc label \( l \), a new token \( l/\theta \) is added to \( p \), i.e., \( M_f^k(p) = M_f^k(p) \cup \{\theta(l/\theta)\} \).

Figure 1 shows an example, where holding, clear, on, handempty, and ontoable are places (circles), stack, unstack, reference, and putdown are arcs (rectangles). The guard condition of \( stack(x,y) \) is \( x/\chi = y \) (enclosed in brackets). An arrow (e.g., from holding to stack) represents a normal arc; a line segment with a small solid diamond on both ends (e.g., between stack and handempty) represents an inhibitor arc. Each arc can be labeled by a tuple of variables and/or constants. If an arc is not labeled (i.e., the arc from handempty to pickup), the default label is the zero-argument tuple \(<\varnothing>\).

We denote a sequence of transition firings as \( M_0^k \{t_1, \theta_1 > M_0^k \ldots \{t_n, \theta_n > M_0^k \} \), or simply \( t_1, \theta_1 \ldots t_n, \theta_n \), where \( M_0^k \) is an initial marking, \( \theta_i \) (1 \( \leq i \leq n \)) is the substitution for firing transition \( t_i \) under \( M_0^k \), and \( M_0^k \) is the resultant marking. A marking \( M_0^k \) is said to be reachable from \( M_0^k \) if there is such a firing sequence that transforms \( M_0^k \) to \( M_0^k \).

B. Test Generation

A test case consists of test input and test oracle. The test input of a (normal) test is a valid firing sequence \( t_1, \theta_1 \ldots t_{n-1}, \theta_{n-1} \ldots t_n, \theta_n \). For each transition \( t(x_1, \ldots, x_m) \) and substitution \( \theta = \{x_1/a_1, \ldots, x_m/a_m\} \), \( t(1 \leq i \leq n) \), also denoted as \( t(a_1, ..., a_m) \), is a test input to the SUT. The input of a robustness test is an invalid firing sequence \( t_{1-1}, \theta_{1-1} \ldots t_n, \theta_n \), where \( t_{1-1} \ldots t_n, \theta_n \) is a valid firing sequence but \( t_{1-1}, \theta_{1-1} \) is not a valid firing. Robustness tests aim at exercising the SUT with invalid inputs so as to verify whether or not the SUT responds correctly. In principle, invalid events can be modeled in function nets by using additional transitions. However, representing various invalid events explicitly in function nets tends to make it difficult to manage test models. Thus, MISTA provides a technique for generating robustness tests automatically.

MISTA provides various methods for automated generation of test inputs from function nets. In this paper, we focus on coverage-based test generation, including the following:

- **State coverage**: A test suite for state coverage covers all markings reachable from the initial markings.
- **Transition coverage**: A test suite for transition coverage covers all transitions reachable from the initial markings.
- **Reachability coverage**: A test suite for reachability coverage covers all state-transitions reachable from the initial markings.

- **Reachability coverage with robustness tests**: A test suite for reachability coverage with robustness tests enhances the test suite for reachability coverage with a robustness test at each reachable marking, if exists.

Coverage-based test suites generated by MISTA are minimized in that: (a) MISTA terminates test generation once the coverage goal is achieved, and (b) MISTA never produces duplicate tests as it organizes all tests in a state transition tree that starts from the initial markings. The reachability coverage test suite subsumes both state coverage test suite and transition coverage test suite. It is subsumed by the suite for reachability coverage with robustness tests.

When generating a normal test input $t_1 \theta_1 \ldots t_n \theta_n$, MISTA allows the oracles to be created in the following ways:

- **States as test oracles**: The test oracles are the resultant markings (states) of respective transition firings, $M_1 \ldots M_n \in M^k$. For each place $p \in P$ and each token $<b_1, \ldots, b_m> \in M_i^k(p)$, $p(b_1, \ldots, b_m)$ is expected to evaluate true in the SUT. So MISTA converts $p(b_1, \ldots, b_m)$ to an assertion after test input $t_i \theta_i$.

- **Postconditions as test oracles**: The test oracles are the effects of respectively transition firings. As mentioned before, the effects of firing $t_i \theta_i$ include: (a) removing tokens from its input places under $M_i \gamma^k$. Each of these tokens should not appear in the resultant state. So MISTA converts it to a negative assertion after test input $t_i \theta_i$. (b) adding new tokens to its output places. We convert each of the new tokens it to an assertion after test input $t_i \theta_i$.

- **States and postconditions as test oracles**: It is a combination of the above two methods. The test oracles include both resultant markings and postconditions of respective transition firings.

For the invalid firing $t_1 \theta_i$ of a robustness test $t_1 \theta_1 \ldots t_n \theta_n$, the test oracle is the marking before $t_1 \theta_i$, meaning that $t_1 \theta_i$ should not change the system state. The generation of executable code for test inputs and test oracles is beyond the scope of this paper. The details can be found in [12][15].

### III. EXPERIMENT SETUP

Our experiments aim to answer the following questions:

**Q1**: What are the fault detection capabilities of coverage-based test suites?

**Q2**: How do robustness tests improve fault detection capability?

**Q3**: How do state-based oracles and postcondition-based oracles contribute differently to fault detection?

Our experiments are based on three Java programs: cruise control, vending machine, and blocks game. Cruise control and vending machine are two popular examples for demonstrating model-based testing. In this paper, the cruise control code originated from [9], and the vending machine code was written by the authors. As shown in Figure 2, the cruise control model is essentially a low-level Petri net. It has no variables, similar to a traditional state machine. The vending machine model is shown in Figure 3. It sells three types of drinks: coffee, soda, and juice. The blocks program originates from the classical planning program in the field of artificial intelligence [10]. In MISTA, it is used as an example for demonstrating the expressive power of function nets. As shown in Figure 1, the blocks model captures both control-related and data-related interactions between four software components: **pickup, putdown, stack** and **unstack**. These interactions can be interpreted in first-order logic and the state transitions depend on various initial states. We believe that it is difficult to specify the behaviors with proposition-level UML state machines.

![Figure 2. Function net for cruise control](image)

![Figure 3. Function net for vending machine](image)
Table I presents the main metrics of the subject programs. Although they are very small in terms of lines of code, they allow us to build systematic test models for the thorough evaluation of testing effectiveness. Our prior work has used MISTA to test large-scale real world applications [12], but focused on system components, rather than entire systems.

For each subject, we conducted the experiment as follows:

1. Build the function net (test model) and MIM (model-implementation mapping for test code generation);
2. Generate test code for each testing strategy under evaluation (i.e., each combination of coverage choices and oracle choices as described in Section II.B);
3. Execute all the test code against the subject program. If there is a failure, then either the subject program or the test code has a problem. In this case, fix the problem in the subject program or repeat steps (1) and (2);
4. Create mutants of the subject program using MuJava [8]. MuJava is a tool for generating Java mutants, executing test code against the mutants, and reporting execution results. Each mutant is a modified version of the subject program with one fault injected. We also created additional mutants by manually injecting the types of faults not covered by MuJava. For example, faults that do not achieve the desired postconditions may be due to missing statements. MuJava does not generate such mutants because removing an arbitrary statement may lead to a syntax error. Table I shows the number of mutants created for each subject program.
5. Execute all the test code in (2) against each mutant using MuJava. A mutant is said to be killed by a test suite if the execution of the test suite reports a failure.

IV. EXPERIMENT RESULTS

A. Coverage-based Tests

We use mutant-killing ratio (i.e., number of killed mutants divided by the total number of mutants) as an indicator of fault detection capability. Figure 4, Figure 5, and Figure 6 show the number of killed mutants and the mutant-killing ratio of each coverage-based test suite in cruise control, vending machine, and blocks. “State”, “Transition”, “Reachability”, and “Reachability+robustness” refer to test suites for state coverage, transition coverage, reachability coverage, and reachability coverage with robustness tests, respectively.

In cruise control and vending machine, the three test oracle methods happen to be identical. Thus, there is only one test suite for each of the coverage criteria in Figure 4 and Figure 5. The overall mutant-killing ratios for all subject programs in Figure 7 indicate that the test suites of state coverage are less capable than the test suites of transition coverage, which are less capable than the test suites of reachability coverage. The state coverage test suite and the transition coverage test suite are both a subset of the reachability coverage test suite. The test suites of reachability coverage with robustness tests are the most powerful – they killed 99.7% of all mutants. They killed all mutants of cruise control and vending machine. The only live mutant of blocks has an endless loop created by MuJava. In this case, MuJava terminated the execution without failure because it used a timeout.

![Figure 4. Effectiveness of coverage-based tests for cruise control](image)

![Figure 5. Effectiveness of coverage tests for vending machine](image)

![Figure 6. Effectiveness of coverage-based tests for blocks](image)

![Figure 7. Overall effectiveness of coverage-based tests](image)

Note that the comparison of test suites for state coverage and transition coverage varies from application to application. In cruise control, transition coverage is more capable (Figure 4). In vending machine, however, state coverage is more capable (Figure 5). They are the same in blocks (Figure 6).
Figure 8. Number of mutants killed per test

Figure 8 shows the average number of mutants killed by each test in coverage-based test suites. Although the test suites for state coverage and transition coverage are less capable than those for reachability coverage and reachability coverage with robustness tests, their individual tests are more effective. In particular, each test in the transition coverage suite killed 15.6 mutants. This has two implications: (1) when testing resource (e.g., time and budget) is very limited, transition and state coverage are better choices. These tests should be conducted first. (2) When the testing process progresses, it gets more and more expensive to find additional faults because more and more tests need to be created and executed.

Figure 9. Lines of test code / mutants killed

Figure 9 shows the average lines of test code for killing mutants. The executable test code is generated automatically. Before test code generation, helper code (e.g., package statement) needs to be provided. In cruise control and blocks, the helper code has only two lines (i.e., package and import statements). In vending machine, the helper code has 16 lines (package and import statements and a method for verifying actual results). These numbers are insignificant compared to the total lines of test code. Table II below shows the lines of test code for reachability coverage and reachability coverage with robustness tests. Figure 9 shows the transition coverage tests used 1.3 lines to kill one mutant, whereas the reachability coverage with robustness tests used 35.7 lines to kill one mutant. Fig. 9 demonstrates the same phenomenon as Fig. 8.

B. Robustness Tests

In MISTA, robustness tests are generated only for reachability coverage. So we evaluate their effectiveness by comparing the test suites for reachability coverage and for reachability coverage with robustness tests, as shown in Table II. Reachability+R denotes reachability coverage with robustness tests and LOC refers to lines of test code. Generally there are more robustness tests than normal tests (in terms of both number of tests and lines of code). The robustness tests contributed to the killing of 37 additional mutants – the test suites for reachability coverage killed none of these mutants.

Table II. Evaluation of Robustness Tests

<table>
<thead>
<tr>
<th>Category</th>
<th>Cruise Control</th>
<th>Vending Machine</th>
<th>Blocks</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests</td>
<td>Reachability</td>
<td>Reachability+R</td>
<td>Increased</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>25</td>
<td>16</td>
<td>234</td>
</tr>
<tr>
<td>Reachability</td>
<td>204</td>
<td>648</td>
<td>444</td>
<td>745</td>
</tr>
<tr>
<td>Increased</td>
<td>217</td>
<td>10,076</td>
<td>1,230</td>
<td>11,523</td>
</tr>
<tr>
<td>Mutants</td>
<td>Reachability</td>
<td>Reachability+R</td>
<td>Increased</td>
<td></td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>113</td>
<td>20</td>
<td>286</td>
</tr>
<tr>
<td>Reachability+R</td>
<td>72</td>
<td>88</td>
<td>12</td>
<td>323</td>
</tr>
<tr>
<td>Increased</td>
<td>511</td>
<td>6,881</td>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10 shows the consequences of using robustness tests – the total number of tests increased by more than 200%, the total lines of code increased by 150%, and the mutant-killing ratio increased by only 13%. This indicates that, when the testing process progresses, much more tests would be required in order to find additional faults. While robustness tests are critical to fault detection, they are also costly because there are often numerous invalid inputs.

C. Test Oracles

In cruise control and vending machine, the three test oracle methods have no difference. In blocks, using the combination of states and postconditions as oracles is the same as using postconditions as oracles (refer to Figure 6) because all mutants killed by the state oracles are also killed by the postcondition oracles. Figure 11 shows the number of killed mutants and the percentage of killed mutants increased by using the postcondition oracles for each of the coverage-based test suites in blocks. The postcondition oracles are more capable than the state oracles because the state oracles do not check negative effects of test actions (i.e., token removal).

Figure 10. Cost effectiveness of robustness tests

Figure 11. Effectiveness of postcondition oracles
D. Threats to Validity

The above experiment results have demonstrated that function testing with Petri nets can be highly effective in fault detection. The main threats to validity are discussed below.

First, the three subjects are very small programs in terms of lines of source code. The benefit is that we can build systematic test models and generate comprehensive tests suites. For complex applications, however, it can be difficult to build systematic test models with function nets. In addition, the methods for generating coverage-based tests may not scale up.

Second, the test oracles in the subject programs can all be represented formally and thus converted into executable assertions. For real-world applications, it is often a challenge to define precise test oracles.

Third, the three subjects are all Java programs. Although Java is a representative object-oriented language, the fault detection capability of a test generation method can depend on particular programming constructs and paradigms.

Finally, the evaluation of fault detection capability is based on fault injection, which is a common approach to the evaluation of testing effectiveness [5]. The mutants were either generated by MuJava or created manually. Although they have covered many types of bugs, they do not necessarily represent all possible faults in real-world software.

V. Related Work

Zhu and He [17] have proposed a methodology for testing high-level Petri nets. The methodology consists of four testing strategies: transition-oriented testing, state-oriented testing, data flow-oriented testing, and specification-oriented testing. Each strategy is defined in terms of an adequacy criterion for selecting test cases and an observation scheme for observing a system’s dynamic behavior during test execution. It is not concerned with how tests can be generated to meet the adequacy criteria. Lucio et al. [7] proposed a semi-automatic approach to test case generation from CO-OPN specifications. CO-OPN is a formal object-oriented specification language based on abstract data types and Petri nets. This approach transforms a CO-OPN specification into a Prolog program for test generation purposes. Lucio developed SATEL [6], a language for expressing test intentions of object-oriented CO-OPN specifications of reactive systems. Test intentions can be used to produce a reasonable or practicable number of test cases, including negative tests (i.e., robustness tests in this paper). Desel et al. [1] have proposed a technique to generate test inputs (initial states) for the simulation of high-level Petri nets. The above work targets the testing and simulation of Petri nets. Wang et al. [11] have proposed class Petri net machines for specifying inter-method interactions within a class and generating method sequences for unit testing. Manual work is needed to make the sequences executable.

VI. Conclusions

We have presented three controlled experiments for evaluating the cost-effectiveness of model-based testing with high-level Petri nets. To the best of our knowledge, this paper is the first empirical study of its kind. The experiment results show that testing with high-level Petri nets can be highly effective in fault detection and that robustness tests and postcondition oracles are both critical to finding faults.

Applying model-based testing to large-scale real-world software may result in very complex test models which yield a large number of test cases. It may be infeasible to generate test suite for reachability coverage with robustness tests, although it is more capable than state and transition coverage test suites. Our future work aims to evaluate the scalability of various test generation strategies in MISTA.

Acknowledgment

This work was supported in part by US NSF under grants CNS 1123220 and CNS 1359590.

References