CompleX-Machine: An Automated Testing Tool Using X-Machine Theory

EKA.Ogunshile

***Abstract*—** This paper is aimed at creating an Automatic Java X-Machine testing tool for software development. The nature of software development is changing. Thus, the type of software testing tools required is also changing. Software is growing increasingly complex and, in part due to commercial impetus for faster software releases with new features and value, increasingly in danger of containing faults. These faults can incur huge cost for software development organisations and users; Cambridge Judge Business School's research estimated the cost of software bugs to the global economy is $312 billion. Beyond the cost, faster software development methodologies and increasing expectations on developers to become testers is driving demand for faster, automated, and effective tools to prevent potential faults as early as possible in the software development lifecycle. Using X-Machine theory, this paper will explore a new tool to address software complexity, changing expectations on developers, faster development pressures and methodologies, with a view to reducing the huge cost of fixing software bugs.

***Keywords*—** Conformance Testing, Finite State Machine, Software Testing, X-Machine.

I. Introduction

FINITE state machines (FSMs) can be used as mathematical representations of the expected states present in a system [1]. FSMs form the basis of development methodologies and system specifications [2], being represented by state diagrams and state transition tables. FSMs can be further extended to use memory, or data, to form X-Machines [1]. These concepts are evolving to enable software tests to be generated against a system specification which can help established the completeness, consistency, and correctness of an implemented system [3]. Given that software testing typically accounts for 50% of the development budget [4], with $2.2 trillion spend on IT annually in 2010 [5], effective testing tools have huge potential to reduce risk and deliver stronger return on investment. Cost tends to be orders of magnitude greater later into the development process. Referencing NIST, a 2008 IBM Whitepaper found the cost of fixing integration errors can be 10x the cost during design and architecture, and twice the cost of during implementation [6]. Morover, most errors tend to be found during the integration phase (see Fig. 1). Beyond financial gain, Wong *et al.* [7] notes that "software faults in safety-critical systems have significant ramifications." The aim of this research is to develop a software testing framework based on X-Machine theory, which can be applied to Java classes to determine conformance to requirements, and address the challenges facing current software testing methods and approaches. These requirements will be provided in the form of XML specifications detailing the expected transitions and states of the Java class being tested. The paper will provide an analysis the challenges that exist for existing testing tool, an overview of X-Machine theory, and review of the state of testing using X-Machine and Finite State Machine theory as it presently exists. The paper will then describe the proposed approach for applying X-Machine theory in a testing tool, and perform a critical review of this approach in the context of software testing.

EKA.Ogunshile is a Senior Lecturer in Computer Science and Head of V& V Research Unit of the Software Engineering Research Group (SERG) at the University of the West of England (UWE), Bristol, UK, BS16 1QY.

II. Related Work

Testing tools uncover errors with varying success and performance and face a range of continuing challenges including:

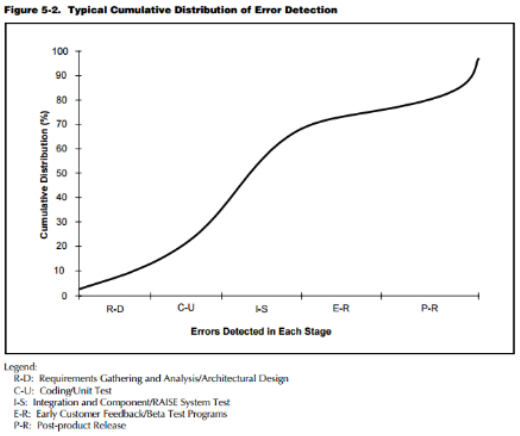

Fig. 1 Typical Cumulative Distribution of Error Detection [8].

- Incomplete error detection
- Software complexity
- Agile Testing
- Automation trade-offs
- Developer adoption

The following sections will elaborate on these problems in the context of an X-Machine based conformance testing solution aimed at preventing errors beyond the software development phase.

III. CURRENT SOFTWARE TESTING CHALLENGES

A. Incomplete error detection

"In general, it is impractical, often impossible, to find all the errors in a program" [9]. For example, unit testing is effective at testing individual methods within a component, but Holcombe [10] notes their ineffectiveness for detecting integration-related errors.

While not an exhaustive test method, NIST suggest falsification testing (also known as conformance testing) as an alternative to prove an implementation under test (IUT) contains errors [5]. The purpose of conformance testing is to determine whether an IUT is correct, complete, and consistent with its specification [11]. If unexpected outputs occur, given the specified test inputs; the IUT does not conform and an error is detected. This method does not guarantee full error coverage, but it has the benefit of significantly reducing the likelihood of present errors because the IUT conforms to its specification. This may prove valuable in preventing a greater number of integration-stage bugs.

B. Software Complexity

Increasing software complexity is causing "more software bugs, which often lead to execution failures with huge losses", particularly because fault localisation can be difficult [7]. Faults can be uncovered by traditional low-level test writing and recording, such as unit testing, which "works fine when there are a small number of tests…but it breaks down as the number of tests grow" [12]. Object-oriented systems are increasing in complexity and scale, making testing a more difficult, costly, and incomplete task. Therefore, a simpler solution is required to validate increasingly complex systems.

C. Agile Testing

Market pressures have shifted the preferred software development lifecycle from rigid waterfall-style methodologies, which enforce staged testing to ensure conformance to specifications [13], to customer-centric agile methodologies which are flexible to changing specifications and emphasise finished products over rigid documentation and testing [14]. Agile software development has grown in popularity three-fold in the past decade [15].

Although unit testing can help prevent errors during development, current integration testing examines "new functionality…once the implementation is done…From a lean perspective, preparing tests afterwards is wasteful." [12]. Manual unit test writing can also prove costly. Future testing tools need to align with the agile development process.

D. Automation trade-offs

Although automated testing has benefits, the trade-off is test customisability and result reliability. Tools such as DSD-Crasher or Daikon are designed to automatically infer a system's intended behaviour and functionality given that "Explicit specifications require significant human effort" [16]. An absolute-automatic approach, however, can result in an unforeseeable number of false positives which the tester cannot rule out. Ruling out these false positives also takes significant human effort. The option of "adapting testing is required to determine effectiveness of test data" [17]. A balance must be struck. "For testing to be efficient, it must be automated as much as possible", but with the necessary tester customisation which delivers appropriate and reliable results [1].

E. Developer adoption

Developers are becoming more responsible for testing. Research underpinning the Agitator testing tool argued it is "difficult for developers to switch modes from development activities – mostly constructive and focused – to testing activities – mostly destructive and exploratory" [18]. Crispin and Gregory [14] identify four developer-related barriers to automated testing including developer overreliance on quality assurance teams to detect faults, the "hump of pain" in learning new tools and code, the fears of testers with weaker programming backgrounds, and habitual comfort in sticking with familiar manual regression testing. Usability, learnability, performance, and overall design must be seamless for developer adoption of new testing tools [18].

IV. FINITE STATE MACHINE & X-MACHINE BACKGROUND

A. Finite State Machines (FSM)

FSMs are a popular, simple way of describing a wide range of systems, including hardware. They have the advantage of being based on simple, dynamic models of computation, are easily represented in diagrams and tables, and are relatively well-known – see working and faulty FSM representations in Fig. 2 and Fig. 3.

Chow's FSM testing method was effective in testing control structure correctness, but for tests involving a "data-manipulation aspect…other testing approaches must be used" [19]. FSM testing also assumes "that not only the specification, but also the implementation can be modelled as an FSM" [11]. Moreover, the FSM characterisation set (Chow's W method, used to distinguish between two pairs of states in machines) only includes the model inputs; not the transition outputs or any values carried in memory. Outputs are observed, which does not necessarily validate a correct transition. Ultimately, this means FSMs face limitations such as modelling non-deterministic behaviour or data manipulation, object-orientation complexity where components may communicate or call values and methods mid-transition, or testing systems that cannot be modelled as FSMs.

B. Stream X-Machines (SXM)

Laycock's Stream X-Machines (also known as extended X-Machines) were aimed at modelling data manipulation in memory [3]. SXM's main advantage is the ability to almost completely model modern Turing machines in a "wide variety of situations in a unified manner and…[control] transformation and refinement of specifications" [2]. Furthermore, SXMs avoid the 'state explosion' problem in Statechart models, where the size of data variables radically exceeds the number of specified states, because the "states of the SXM equivalent of a state diagram coincide with the original states" [1].

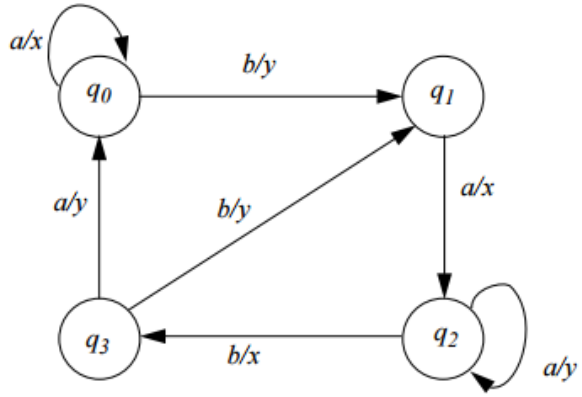

Fig. 2 A simple X-Machine [10]

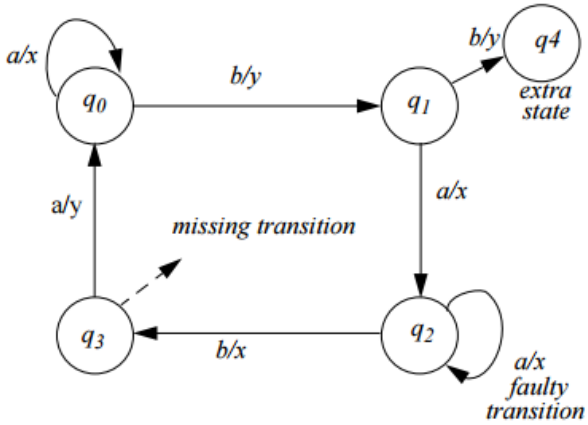

Fig. 3 A faulty version of an X machine with an extra state, a missing transition and faulty transition label [10]

C.Communicating X-Machines

Beyond simple SXM modelling, complex system "modelling of concurrency and communication is made possible by invoking net-like models or by using machine product constructions of a suitable type" [2]. Communicating X-Machines have been proposed to model more complex systems, based upon a single CSXMS or a CSXMS composed of from several component CSXMSs which "give the software designer the freedom to choose the level of detail at which to apply the X-Machine model to any particular software system" [20]. The CSXMS enables the modelling of complex system behaviour such as "such as determinism, minimality and output distinguishability". [20]. Balanescu *et al.'s* [20] port-based CSXMS was "not the only possible formulation". While Barnard *et al.* [21] for example, implemented a port-based system, Simons *et al.* [22] developed an Object Machine which better "describes the state changes and responses of an object triggered by the reception of message request", as in object-oriented systems.

V.OUR SOLUTION: AN SXM-BASED CONFORMANCE TESTING

Our test draws Aguado and Cowling's conformance test using SXMs to "determine whether an IUT conforms (is equivalent) to its specification [23].

This technique aims to find the same faults as in FSM testing, namely:

- Missing states
- Extra states
- Missing Transitions
- Extra Transitions
- Mis-directed Transitions
- Transitions with faulty-functions (input-output).

"A significant advantage is that the same approach can be applied to each component of the systems if each [component] is specified as an SXM" [23]. Holcombe and Ipate's [24] 'reductionist' approach better suits agile development where systems are assembled from components. Their approach involves producing a test regime that completely reduces "the test problem for the system to one of looking at the test problem for the components or reduced parts" Holcombe [10]; as per the agile process. This enables greater system-level conformance testing throughout development to detect errors earlier in the process [10]. This approach should also help address the issue of increasing software complexity at a much more granular level of computational testing.

Conformance testing should scale to deliver effective test results with larger and more complex systems, given "the larger and more varied the set of inputs is, the more confidence can be placed in an implementation whose testing generates no errors" [5]. Conformance testing of individual components, though "not a guarantee for interoperability, it is an essential step towards achieving interoperability" [5]. If every component's functionality conforms to specifications, the fully integrated system is more likely to successfully conform to specifications. A CSXMS conformance tool would arguably be the next step for a fuller integration conformance test during development [20].

Attractiveness for developers, automation and customisability depends to a large extent on the implementation of the testing tool. However, X-Machine specification-based theory should enable a substantially automated conformance testing tool with flexibility to accommodate a wider variety of specified system designs and implementations.

VI. METHOD / FRAMEWORK

The first iteration of the testing framework involves testing for missing transitions, extra transitions, and misdirected transitions. By testing in these areas, the framework will be able to test for conformance to specifications, both in terms of adherence to system designs and functional expectations of a class.

A. Specifications

The expected behaviour of the system is captured by specifying the transitions expected in the system with any

inputs required (i.e. the methods or functions of the class), along with the state the system will initially be in, and the state expected at system completion. States are defined by values set in class variables.

This can be represented in a similar way to state transition tables used with FSMs, with some modification to represent the X-Machine, such as the addition of a column to represent the transition, and the name of the class variable with its value in each state. Table 1 shows an example table representing a traffic light system.

Table 1

EXAMPLE STATE TRANSITION TABLE FOR A TRAFFIC LIGHT SYSTEM

Starting State	**Transition**	**Finishing State**
color = green	prepareToStop()	color = amber
color = amber	stop()	Color = red
color = red	prepareToGo()	color = amber
color = amber	go()	color = green

B.Missing Transitions

To test for missing transitions, the expected transitions will be extracted from the specification, and compared to the methods or functions present in the class. Any transitions present in the specification, yet not present in the system, will be classed as missing.

Using Table 1 as a specification, a system with methods called prepareToStop(), stop(), pepareToGo(), and go() will pass all the tests. However, a system missing the prepareToStop() method would be a partial failure, as it does not conform to the specification. A system with none of the transitions specified would be a total failure.

C.Extra Transitions

Extra transitions will be methods or functions present in a system that are not present in the specification. Again, transitions are extracted from the specification and compared to the methods or functions present in the system. Any methods or functions that are not present in the specification will be classed as extra transitions.

For the traffic light example, a class only containing the four methods in the specification will pass the test. However, if the class contains methods not specified in the transition, the test will fail as it does not conform to the specifications.

D.Misdirected Transitions

Misdirected transitions are those which do not put the system in the expected state at the end of the transition. States are defined by class variables and their values. In the example specification in Table 1, the first state defined is color = green. This expects the system to have a class variable called “color”, which is given the value “green”. To run this test, the system is placed in the starting state for a transition. The transition is then run on the system, and the class variables tested to ensure they are as specified in the finishing state. This is repeated for all transitions in the specification.

Using the traffic light example, for each transition test, the color variable is assigned the value specified in the starting state. After calling the method described by the transition, the color variable is interrogated to ensure the value is that specified in the finishing state. If the variable contains an incorrect value, the test is failed.

VII. THE COMPLEX-MACHINE TOOL

The CompleX-Machine Tool has been developed in Java to accept specifications and system files to test for missing transitions, extra transitions, and misdirected transitions. It utilizes JavaParser and Javassist to manage the parsing, interrogating, and running of the system files.

A. Specifications

Specifications are parsed from an XML file which is then converted in a model as described in Fig. 4. An XML Schema representing a well-formed specification file can be found in Appendix 1. Tests are generated based on the specification.

B.Missing Transitions

Missing transitions are identified by comparing a list of transitions from the specification to a list of methods extracted from the submitted system file, as parsed using JavaParser. Any transitions present in the specification but not present in the system are classed as failed tests. Transitions and methods are compared using the name of the transition or method, along with parameters required for the transition or method. For example, a specification requiring a transition called changeColor with a parameter of type String called color, will look for a method signature matching changeColor(String color). For the purposes of this test, return types, access modifiers, and throws declarations are ignored.

C.Extra Transitions

Extra transitions are tested in a similar way to missing transitions, the difference being that that methods present in the system, yet not present in the specification, are classed as failed tests.

D.Misdirected Transitions

To test misdirected transitions, the system file is instantiated using Javassist so that the methods can be called and the class variables interrogated.

Again, for the purposes of this tool, access modifiers are ignored. Once the class has been instantiated, for each transition, the class variables are set to those in the specification.

The method is called on the object, and the class variables tested for equality to those defined in the finishing state of the specification. If the class variables do no match those specified, the test is classed as failed.

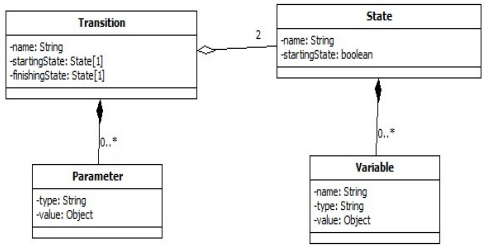

Fig. 4 System model representing the test specifications

VIII. VALIDATION

The tool has been validated for missing transitions, extra transitions, and misdirected transitions using test files representing a traffic light system, and a string manipulator. The XML specifications can be found in Appendix 2 and Appendix 3 respectively.

```
public class TrafficLight {
    public java.lang.String color;

    public void prepareToStop() {
        color = "amber";
    }

    public void stop() {
        color = "red";
    }

    public void prepareToGo() {
        color = "amber";
    }

    public void go() {
        color = "green";
    }
}
```

Fig. 5 Java file conforming to the traffic light specification

```
public class TrafficLight {
    public java.lang.String color;

    public void prepareToGo() {
    }

    public void go() {
    }
}
```

Fig. 6 Java class with some methods missing compared to the traffic light specification

```
public class TrafficLight {
    public java.lang.String color;
}
```

Fig. 7 Java class with all methods missing compared to the traffic light specification

A. Missing Transitions

To ensure that the tool could identify transitions present in both the specification and system file, a java file was created containing all the transitions expected of the Traffic Light specification (see Fig. 5). Upon running the test, the expected transitions were all present and all tests marked as passed.

Two more tests were performed, one with two missing transitions (see Fig. 6) and the other with no transitions in the system (see Fig. 7). The transitions that had been removed from the java file were classed as failed tests, while those that remained were passed.

By running these tests, we have been able to show that the tool can recognise which transitions should be present in the system and identify those that are missing.

B. Extra Transitions

The first part of this validation was similar to that of missing transitions. The file shown in Fig. 5 was again run against the traffic light specification to ensure that no transitions were marked as extra if they were present in the specification. This test was successful. From there, a further java file was created containing a method which was not defined in the specification (see Fig. 8). The tool identified the extraTransition() method as not being defined in the specification and correctly failed that test. The correct transitions all passed as expected.

```
public class TrafficLight {
    public String color;

    public void prepareToStop() {
    }

    public void stop() {
    }

    public void prepareToGo() {
    }

    public void go() {
    }

    public void extraTransition() {
    }
}
```

Fig. 8 Java file containing a method not defined in the specification

C.Misdirected Transitions

The initial test of misdirected transitions used the traffic light specification and the java file used in both the missing and extra transitions tests. As this is a correct representation according to the specification, the transition tests all passed as expected. In order to ensure that misdirected transitions could be identified correctly, the java files in Fig. 9 and Fig. 10 were tested against the traffic light specification. The partially misdirected file successfully passed the two transitions which ended in the correct states, while failing those that did not. The file with completely incorrect transitions failed all the tests as expected. Further validation was performed using a file with a single method, as specified in Appendix 3. In this system, the finishing state is dependent on the starting state, rather than the transition called. The java file used for this test can be seen in Fig.11. The tool could call the method and identify that the final state was correct, based on the starting state. As such, the tool can test for functional conformance, as well as design conformance.

```
public class TrafficLight {
    public java.lang.String color;

    public void prepareToStop() {
        color = "amber";
    }

    public void stop() {
        color = "blue";
    }

    public void prepareToGo() {
        color = "amber";
    }

    public void go() {
        color = "amber";
    }
}
```

Fig. 9 Java file with some transitions misdirected

```
public class TrafficLight {
    public java.lang.String color;

    public void prepareToStop() {
        color = "red";
    }

    public void stop() {
        color = "green";
    }

    public void prepareToGo() {
        color = "red";
    }

    public void go() {
        color = "amber";
    }
}
```

Fig. 10 Java file with all transitions misdirected

```
public class SingleTransition {
    java.lang.String value = "a";

    public void convert() {
        if(value.length() == 1) {
            value = value + "a";
        } else {
            value = value.substring(0, 1);
        }
    }
}
```

Fig. 11 Java File with a single transition

IX. Conclusion

The aim of this paper was to create a software testing tool using X-machine theory and address the challenges and limitations for current testing methods. Early research into testing using X-Machine theory identified that testing for states and transitions was key. The CompleX-Machine uses X-Machine theory to test for extra transitions, missing transitions, and misdirected transitions. These tests have been validated with test cases using specific specification and system files where if passed are completing the transition tests correctly. Currently the system is not testing for states, however, this does not inhibit the application from displaying the benefits of testing using X-Machine theory. Using conformance testing reduces the possibility of errors as it is conforming to the specification. In the future, the state testing feature would need to be added to the application to completely test a system following the basis of the X-Machine model. Tests would be like the transition tests, looking for missing states or extra states. Additionally, allowing users to use different types of specification inputs would make the application far more flexible. For example, allowing users to upload a FSM or X-Machine diagram as the specification input, users could then draw out what they are visualizing.

Appendix 1

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="specification">
  <xs:complexType>
   <xs:element name="transitions">
    <xs:complexType>
     <xs:element name="transition">
      <xs:complexType>
       <xs:all>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="startingState" maxOccurs="1"
minOccurs="1">
         <xs:complexType>
          <xs:group ref="state"/>
         </xs:complexType>
```

```
            </xs:element>
            <xs:element      name="finishingState"      maxOccurs="1"
minOccurs="1">
              <xs:complexType>
                <xs:group                                    ref="state"/>
              </xs:complexType>
            </xs:element>
            <xs:element       name="parameters"       maxOccurs="1"
minOccurs="0">
              <xs:complexType>
                <xs:element                                name="parameter">
                  <xs:complexType>
                    <xs:element      name="name"      type="xs:string"/>
                    <xs:element       name="type"       type="xs:string"/>
                  </xs:complexType>
                </xs:element>
              </xs:complexType>
            </xs:element>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:complexType>
  </xs:element>
 </xs:element>

 <xs:group                                                  name="state">
  <xs:all>
    <xs:element              name="name"              type="xs:string"/>
    <xs:element                                       name="variables">
      <xs:complexType>
        <xs:element                                    name="variable">
          <xs:complexType>
            <xs:element          name="name"          type="xs:string"/>
            <xs:element           name="type"          type="xs:string"/>
            <xs:element          name="value"          type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:complexType>
    </xs:element>
    <xs:element       name="startingState"       type="xs:boolean"/>
  </xs:all>
 </xs:group>
</xs:schema>
```

X. Appendix 2

```
<specification xsi:noNamespaceSchemaLocation="specification.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <transitions>
    <transition>
      <name>prepareToStop</name>
      <startingState>
        <name>green</name>
        <variables>
          <variable>
            <name>color</name>
            <type>java.lang.String</type>
            <value>green</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </startingState>
      <finishingState>
        <name>amber</name>
        <variables>
          <variable>
            <name>color</name>
            <type>java.lang.String</type>
            <value>amber</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </finishingState>
    </transition>
    <transition>
      <name>stop</name>
      <startingState>
        <name>amber</name>
        <variables>
          <variable>
            <name>color</name>
            <type>java.lang.String</type>
            <value>amber</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </startingState>
      <finishingState>
        <name>red</name>
        <variables>
          <variable>
            <name>color</name>
            <type>java.lang.String</type>
            <value>red</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </finishingState>
    </transition>
    <transition>
      <name>prepareToGo</name>
      <startingState>
        <name>red</name>
        <variables>
          <variable>
            <name>color</name>
            <type>java.lang.String</type>
            <value>red</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </startingState>
      <finishingState>
        <name>amber</name>
        <variables>
          <variable>
            <name>color</name>
            <type>java.lang.String</type>
            <value>amber</value>
          </variable>
        </variables>
```

```
        <startingState>
          false
        </startingState>
      </finishingState>
    </transition>
    <transition>
      <name>go</name>
      <startingState>
        <name>amber</name>
        <variables>
          <variable>
            <name>color</name>
            <type>java.lang.String</type>
            <value>amber</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </startingState>
      <finishingState>
        <name>green</name>
        <variables>
          <variable>
            <name>color</name>
            <type>java.lang.String</type>
            <value>green</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </finishingState>
    </transition>
  </transitions>
</specification>
```

XI. APPENDIX 3

```
<specification xsi:noNamespaceSchemaLocation="specification.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <transitions>
    <transition>
      <name>convert</name>
      <startingState>
        <name>odd</name>
        <variables>
          <variable>
            <name>value</name>
            <type>java.lang.String</type>
            <value>a</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </startingState>
      <finishingState>
        <name>even</name>
        <variables>
          <variable>
            <name>value</name>
            <type>java.lang.String</type>
            <value>aa</value>
          </variable>
        </variables>
```

```
        <startingState>
          false
        </startingState>
      </finishingState>
    </transition>
    <transition>
      <name>convert</name>
      <startingState>
        <name>even</name>
        <variables>
          <variable>
            <name>value</name>
            <type>java.lang.String</type>
            <value>aa</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </startingState>
      <finishingState>
        <name>odd</name>
        <variables>
          <variable>
            <name>value</name>
            <type>java.lang.String</type>
            <value>a</value>
          </variable>
        </variables>
        <startingState>
          false
        </startingState>
      </finishingState>
    </transition>
  </transitions>
</specificatio
```

REFERENCES

[1] F. Ipate, "Class testing from state diagrams using stream X-machine based methods", in *18th Australian Software Engineering Conference 2007 (ASWEC'07)*, Melbourne, 2007, pp. 245-254.

[2] M. Holcombe, "X-machines as a basis for dynamic system specification", *Software Engineering Journal*, vol. 3, no. 2, p. 69, 1988.

[3] G. Laycock, "The Theory and Practice of Specification Based Software Testing", Ph.D, University of Sheffield, 1993.

[4] H. Tahbildar, P. Borbora and K. G.P, "Teaching Automated Test Data Generation Tools for C, C++ , and Java Programs", *International Journal of Computer Science and Information Technology*, vol. 5, no. 1, pp. 181-195, 2013.

[5] "What is this thing called Conformance?", *NIST*, 2010. [Online]. Available: https://www.nist.gov/itl/ssd/information-systems-group/what-thing-called-conformance. [Accessed: 19- Apr- 2017].

[6] *Minimizing code defects to improve software quality and lower development costs*, 1st ed. New York: IBM Corporation, 2017.

[7] W. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, "A Survey on Software Fault Localization", *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707-740, 2016.

[8] *The Economic Impacts of Inadequate Infrastructure for Software Testing*, 1st ed. National Institute of Standards and Technology, 2002.

[9] G. Myers, *The Art of Software Testing*, 1st ed. New Jersey: John Wiley and Sons, Inc., 2004, p.9.

[10] M. Holcombe, "Testing, testing, testing!", in *Correct Systems – Building Business Process Solutions*, 1st ed., M. Holcombe and F. Ipate, Ed. London: Springer-Verlag, 1998, pp. 61-92.

[11] K. EI-Fakih, N. Yevtushenko and G. Bochmann, "FSM-based incremental conformance testing methods", *IEEE Transactions on Software Engineering*, vol. 30, no. 7, pp. 425-436, 2004.

[12] R. Mugridge, R. Utting and D. Streader, "Evolving Web-Based Test Automation into Agile Business Specifications", *Future Internet*, vol. 3, no. 4, pp. 159-174, 2011.
[13] A. Ahmed, "Software Requirements Management", in *Software Project Management: A Process-Driven Approach*, 1st ed., A. Amed, Ed. Florida: Taylor and Francis Group, 2017, pp. 145-157.
[14] L. Crispin and J. Gregory, "Automation", in *Agile Testing: A Practical Guide for Testers and Agile Teams*, 1st ed., L. Crispin and J. Gregory, Ed. Boston: Pearson Education, 2009, pp. 255-271.
[15] VersionOne Inc, "10th Annual State of Agile Report", VersionOne Inc, 2016.
[16] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A hybrid analysis tool for bug finding", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 2, pp. 245-254, 2006.
[17] H. Tahbildar and B. Kalita, "Automated Software Test Data Generation: Direction of Research", *International Journal of Computer Science & Engineering Survey*, vol. 2, no. 1, pp. 99-120, 2011.
[18] M. Boshernitsan, M. Doong and A. Savoia, "From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing", in *Fifth International Symposium on Software Testing and Analysis*, Portland, 2006.
[19] T. Chow, "Testing Software Design Modeled by Finite-State Machines", *IEEE Transactions on Software Engineering*, vol. -4, no. 3, pp. 178-187, 1978.
[20] Balanescu, T., Cowling, A. J., Georgescu, H., Gheorghe, M., Holcombe, M. and Vertan, C, "Communicating stream X-Machines systems are no more than X-Machines", in *Journal of Universal Computer Science*, vol. 5, no. 9, pp. 494-507, 1999.
[21] J. Barnard, J. Whitworth, and M. Woodward, "Communicating X-Machines", in *Information and Software Technology*, vol. 38, no. 6, pp. 401-407, 1996
[22] A. Simons, K. Bogdanov and M. Holcombe, "Complete functional testing using Object Machines", University of Sheffield, Sheffield, 2017.
[23] J. Aguado and A.J. Cowling, "Foundations of the x-machine theory for testing", University of Sheffield, Sheiffeild, 2017
[24] M. Holcombe, P. Thomas and R. Paul, *Correct Systems*, 1st ed. London: Springer London, 1998, pp. 61-92.

EKA.Ogunshile received the BEng(Hons), MSc(Eng), and Ph.D. degrees in Computer Science from the University of Sheffield, UK, in 2003, 2005, and 2011, respectively. Currently he is a Senior Lecturer in Computer Science at the University of the West of England, Bristol, UK. His research lies broadly in Software Engineering, Model-Driven Engineering, Object-oriented Programming, Verification & Testing and Cloud Computing.