An algorithm for implementing and generating test cases from a minimal stream X-Machine

Authors Name/s per 1st Affiliation *(Author)*
dept. name of organization
City, Country
Email: name@xyz.com

Authors Name/s per 2nd Affiliation *(Author)*
dept. name of organization
City, Country
Email: name@xyz.com

***Abstract*—The rapid change of requirements has made software more complex and harder to maintain. Software testing tools play an important role in the Software Development Life Cycle. However, many technology companies have employed fast paced development of software to meet the demands of their markets. Early released software tends to contain serious bugs and errors because they have not been tested properly during the testing process. Many current available testing tools are capable of detecting these faults; however, some of which are either outdated or cannot catch up with the constantly changing demands. Also, these testing tools concentrate more on generating efficient test sets while ignoring their effectiveness. This paper introduces an algorithm called T-SXM which is developed using the Java programming language based on the stream X-Machines theory which is an intuitive and powerful technique for modelling real world problems formally. The T-SXM algorithm is designed to provide developers with a tool to test the correctness of their implemented systems which can potentially resolve the problem of effectiveness that the other testing tools are facing.**

Keywords-component; T-SXM; testing strategy; transform; model; specification; correctness; effectiveness

I. INTRODUCTION

In Software Development Life Cycle (SDLC), software testing plays an essential role with the intention of finding out all the bugs in the program ([1], [2], [3], [4]). Initially, the tester does the manual testing - a prolonged and tedious process that requires more time to execute [5]. Subsequently, automated testing is carried out in the software development process. However, according to [6], it is still a challenge to have a fully automated software. There are two types of software testing: Black box testing, also known as functional testing ([7], [8]), and White box testing. In White box testing, search-based testing (SBT) and model-based testing (MBT) are both relevant approaches [9] in which the source code is necessary. In SBT, early fault detection in the specification is not possible ([10], [11]). MBT, on the other hand, is essential in finding errors earlier than SBT ([7], [12], [13]).

For object-oriented systems, especially the Java programming language (JPL), the use of formal methods to automatically generate test suites have been extensively studied ([14], [15], [16], [17], [18]). However, most of these approaches concentrated on generating efficient test sets and nothing is said about their effectiveness. Therefore, it implies that these approaches focus more on testing how well the system's functions perform, but not on the correctness of the whole system.

Furthermore, according to Kumar et al. [19], software faults can arise in any phase of the SDLC including requirements gathering and specifications, designs, code, or maintenance. Therefore, the lure of early fault detection and improving the system quality has attracted a considerable attention from research community [19] with a wide range of statistical and machine learning techniques that has been applied to construct the fault prediction models ([20], [21], [22]). Furthermore, the availability of open-source and publicly accessible software fault dataset repositories such as NASA Metrics Data Program and PROMIS has enabled researchers to further investigate on this new area of application [19]. The early fault detection can help software testers or developers to narrow down the risky areas to optimize and prioritize the testing efforts and resources.

In order to resolve the problem of effectiveness and correctness, this research aims to develop an algorithm that can transform an implemented system, especially systems written in the Java programming language, into a stream X-Machine model based on its specifications so that the system can be tested as a whole. This approach can ensure the system behaves correctly and the faults in the system can be discovered early in the development stage.

This paper is structured as follows; section II provides research and background information about the concepts of X-Machines and stream X-Machines. Also, within this section, different software testing tools are reviewed to analyse their strengths and weaknesses. Section III is where we propose the T-SXM algorithm. Section IV describes how the T-SXM algorithm is designed. Section V is where a case study is provided to demonstrate how the T-SXM algorithm works in practice. Section VI is the evaluation of the proposed solution. The future work and conclusion sections are stated the end of this paper.

II. BACKGROUND

A. The theory of X-Machine and stream X-Machine

1) Concepts

According to Ipate et al. [23], the problem of test effectiveness mentioned in the Introduction section can be best addressed if the generated test set is able to find *all faults* in the system under test (SUT). In order to do this, algebraic objects (one is the specification, and the other is the

implementation) need considering, each of which is characterized by an input/output behaviour. If these objects behave in the same way for any input in the test set, they will coincide with any input in the domain. Therefore, it can be concluded that the specification and the implementation behave identically when supplied with the inputs in the test set. This approach has been applied in generating test set for software modelled by *finite state machines* (FSM) (i.e. [24], [25], [26], [27]). Here, the control aspects of the software are assumed to be separated from the system data and can be modelled by an FSM. However, it is difficult to completely separate the system controls from their data, therefore, a more complex specification model that can integrate both of these two aspects is needed.

Such a model is the X-Machine, an enhanced version of FSMs, with data structures and processing functions. In general, an X-Machine is similar to an FSM but with a basic data set, X, and a set of processing functions, Φ, which operate on X. Each arrow in the FSM diagram is then labelled by a function from Φ. In the machine, the sequences of state transitions determine the processing of the data set and therefore, the function or relation is computed. An X-Machine can potentially model very general systems as the data set X can contain information about the system internal memory as well as different output behaviours. Since introduced in 1974 by Eilenberg [28], the X-Machine has been demonstrated to be an intuitive and easy to use model through a number of investigations and research ([27], [29], [30]).

According to Ipate et al. [23], a number of classes of X-Machines have been identified and studied by restricting on the underlying data set X, and the set of processing functions Φ, of the machines. Among these classes, the stream X-Machine (SXM) has received the most attention. The SXM is supposed to resolve a problem that exists in the original X-Machine which is the lack of the ability to process sequences of inputs and outputs. Also, the SXM models have been modified so that they can simulate real world dynamic environments such as object-oriented specifications or deterministic or non-deterministic systems [31].

In SXMs, input and output sets are streams of symbols. The input stream is processed in symbol by symbol from left to right, producing, in turn, a stream of outputs and a regularly updated internal memory. Each processing function processes a memory value and an input to produce an output and a new memory value [23].

An SXM is a tuple $\boldsymbol{Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)}$ where:

- Σ is a finite set of input symbols
- Γ is a finite set of output symbols
- Q is a finite set of states
- M is a (possibly) infinite set called memory
- Φ is a finite set of partial functions φ (processing functions) that map memory-input pairs to output-memory pairs, $\varphi: M \times \Sigma \rightarrow \Gamma \times M$
- F is the next-state partial function, $F: Q \times \Phi \rightarrow Q$
- $q_0 \in Q$ and $m_0 \in M$ are the initial state and initial memory respectively.

For an SXM Z as defined above, the associated FSM, which is also called the associated finite state automaton (FA), is: $\boldsymbol{Az = (\Phi, Q, F, q_0)}$.

Figure 1 is an example of a stream X-Machine model. used to construct the T-SXM.

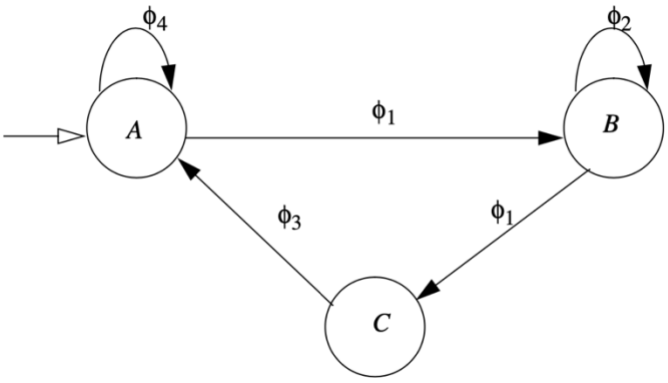

Figure 1. A three-state stream X-Machine

2) Minimal stream X-Machine

According to Ipate [38], an essential problem that arises in the specification process is the capability of finding a "minimal" specification for a required functionality. This problem has been investigated for the FSMs and several types of minimality have been identified such as minimal FSMs, minimal cover-automata, minimal sequential machines, etc.

In this paper, we will focus on investigating the minimality issue in the context of stream X-Machines and only addressing the *state-minimal* SXM with respect to Φ. A state-minimal SXM is machine with as few states and arcs as possible for a given behaviour. In short, *an SXM is minimal if and only if it is accessible and reduced* [38]. In order to make a specified machine minimal, we must show its uniqueness up to a re-labelling of its state space. This means that all the inaccessible states can be removed along with all arcs emerge from or arrive to them without affecting the function computed by the machine. Making an SXM minimal is highly important when it comes to testing because it helps to construct the "smallest" model for a required functionality using the set of processing functions Φ [38].

B. Software testing tools

Chin et al. [32] defined software testing as a process of ensuring a program is free of bugs and performs its intended functions. Software testing tools are used to test systems during SDLC and in post release. The range of available testing tools is vast, each of which has its own advantages and disadvantages. Testing tools help development teams evaluate the correctness, consistency, completeness and quality of a system. In this section, we will be reviewing some of the current software testing tools to analyse their advantages and disadvantages.

1) JUnit

According to Louridas [33], JUnit is a Java-based framework for writing and running tests. Developers use this framework to ensure that the software works as expected and the changes in the implementation do not negatively affect the other features in the SUT.

TABLE I. ADVANTAGES AND DISADVANTAGES OF JUNIT

Advantages	Disadvantages
Can ensure full test coverage by inferring required test cases.	There are no options to write assertions [34].
Useful when the source code is subjected to rapid changes as test cases are generated and inferred automatically.	Has difficulties in performing exhaustive tests [34].
Can test compiled codes.	Cannot tell if a raised exception is an actual fault from the program or its expected behaviour [35].
Can perform systematic testing by executing all transitions between states [34].	
Can be run as a standalone application or an added plugin in IDEs by calling its provided API [35].	

2) JWalk

JWalk is a testing tool that utilizes the lazy specification which is derived from the source, analysis and hints provided by the developers.

TABLE II. ADVANTAGES AND DISADVANTAGES OF JWALK

Advantages	Disadvantages
Can ensure the function output is as expected [33].	Focuses primarily on individual function testing rather than whole system testing.
Can verify if the system performs as expected with minimal disruptions when changes happen.	Operates on the source code, not on the production system.
Has Test Driven Development support and can relatively fit into the SDLC.	Requires some extra libraries such as Mockito, PowerMock, and JUnit to effectively test more complex systems.
Can be used at the integration testing stage.	Can have side effects for other tests when database testing is included [36].
	Is not human readable to non-technical people. However, this can be resolved with the help of external libraries such as Hamcrest.
	Does not automate documentation fully.
	Depends on developers' discretion for what should be tested.

3) JCrasher

JCrasher is a tool that is used to test the robustness of Java systems. While other tools such as JUnit concentrate on testing the system's functionalities and expected outputs from the functions, according to Csallner and Smaragdakis [37], JCrasher generates random, well-formed inputs to test the system in order to identify crashes.

TABLE III. ADVANTAGES AND DISADVANTAGES OF JCRASHER

Advantages	Disadvantages
Can collaborate well with JUnit when it is able to produce test files for JUnit [37].	Requires classes to be set manually to initialize test sets, which increases the time and effort to use the tool when comparing with other tools which automatically identify classes.
Can be used as an add-on for Eclipse IDE to help it fit into the SDLC.	Can potentially generate false positives [37] which leads to an increase in the amount of effort required to interpret the results since these will need to be distinguished from actual system bugs.
Can be able to distinguish between actual program bugs and violations of program preconditions [37].	
Can be fully automated which helps reduce testing time for developers.	

4) Summary

As can be seen in the analysis, the three reviewed tools have their own strengths and weaknesses. However, all of them tend to focus on testing individual functions rather than testing the SUT as a whole. This implies that they cannot tell if an SUT behaves correctly as it is expected to. Also, these tools have difficulties in producing exhaustive tests which leads to an increase in the amount of time and effort for the developers to perform some extra manual interpretations. Finally, these tools do not strictly follow the X-Machine theory.

III. PROPOSED SOLUTION

This paper proposes an algorithm called T-SXM to develop a generic model of stream X-Machine that can be used to test an SUT using its formal specification. With T-SXM, the developers can:

- Determine the formal specification (also called the specification SXM) of the SUT using the Java programming language (see example in section V.A).
- Test the correctness of the SUT by determining different paths for the specification SXM to execute. By doing this, the developers can have a good overview of how well the system is handling unexpected inputs. Once the SXM has processed all the input values, it will produce a sequence of outputs that correspond to the outputs of individual functions which helps the developers to track the behaviours of the functions within the system (see example in section V.B).

The T-SXM will be suited for agile developers as they will benefit with:

- The ability to input specifications to inform the algorithm about the expected states and transitions.
- The ability to test the SUT in different scenarios by simply changing the transition diagram or the input values.

- The ability to test the system as a whole to ensure the correctness is maintained while still knowing which individual function behaves as expected.

IV. THE T-SXM ALGORITHM

A. The package and class diagram

Figure 2 depicts the class diagram which is the design solution for the T-SXM algorithm, where:

- ***SXM***: the class containing the generic SXM model.
- ***InputSequence***: input alphabet of the SXM.
- ***OutputSequence***: output alphabet of the SXM.
- ***ErrorCode***: a Java enum object containing a number of errors that can happen while the SXM is running. The error codes are as follows:
  - ***None***: the SXM has no error while running.
  - ***NoProcessingFunctionFound***: the SXM tries to execute a processing function that does not exist in the specification.
  - ***NoTransitionFound***: the SXM tries to move a new state from a given state and processing function, but it cannot find any matched transition.
  - ***IllegalArgumentForProcessingFunction***: the SXM tries to execute a processing function with invalid arguments.
  - ***ErrorExecutingProcessingFunction***: the SXM tries to execute a processing function but error happens.
- ***States***: state set of the SXM.
- ***Memory***: memory of the SXM.
- ***ProcessingFunctions***: processing function set of the SXM. Each processing function consists of one actual method from the system and the context of the class to which the method belongs.
- ***Transitions***: transition set of the SXM which maps one state and processing function to the next state.

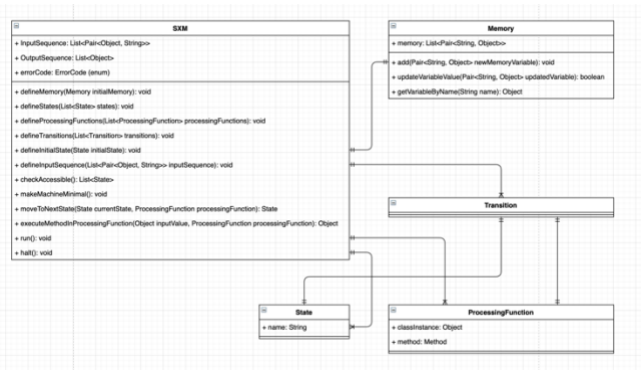

Figure 4. T-SXM class diagram

B. The related algorithms

This section demonstrates the related algorithms that are used to construct the T-SXM.

1) Algorithm for checking if an SXM is accessible

As discussed in section II.A.2, a minimal SXM must be accessible, which means all the states in the machine can be reached from the initial state. Therefore, it is essential to have a dedicated algorithm to check if all the states in a specified SXM are reachable.

```
Input: None
Output: List of inaccessible states

    Initialize a list of accessible states called "accessibleStates" with 1 element – the initial state
    Initialize a list of inaccessible states called "inaccessibleStates" that is equal the initial list of states

    Repeat while the size of accessibleStates > 0:
        countChecked = 0;

        Initialize an empty list called statesToBeCheckedNextLoop

        For state in accessibleStates:
            If state is in inaccessibleStates, then:
                For transition in Transitions:
                    If transition starts from state, then:
                        Add state to statesToBeCheckedNextLoop

                Remove state from inaccessibleStates

            Else, then:
                countChecked += 1;

        If size of statesToBeCheckedNextLoop > 0, then:
            Add all statesToBeCheckedNextLoop to accessibleStates

        If countChecked = size of accessibleStates and size of inaccessibleStates > 0, then:
            Stop the loop


    Return inaccessibleStates
```

Figure 2. Algorithm for checking if an SXM is accessible

2) Algorithm for making an SXM minimal

As stated in section II.A.2, it is essential to check the minimality of the SXM; therefore, we design a specific algorithm to turn a non-minimal SXM into a minimal SXM.

```
Input: None
Output: None

    inaccessibleStates = checkAccessible();

    If size of inaccessibleStates > 0, then:
        For inaccessibleState in inaccessibleStates:
            Remove inaccessibleState from States
            Remove all transitions containing inaccessibleState in Transitions
```

Figure 3. Algorithm for making an SXM minimal

3) Algorithm for executing a processing function

The T-SXM algorithm allows developers to define the system specification in the Java programming language. Therefore, the system's functions will be executed when the machine runs.

```
Input: inputValue, processingFunction
Output: returned value from the processing function

    method = obtained method from processingFunction

    context = obtained context from processingFunction

    If inputValue != number of parameters required by method, then:
        Return IllegalArgumentForProcessingFunction

    If inputValue's types are not matched with parameters' types in method, then:
        Return IllegalArgumentForProcessingFunction

    returnedValue = Invoke method by passing context and inputValue and obtain returned value

    Return returnedValue
```

Figure 5. Algorithm for executing a processing function

4) Algorithm for the next-state function

The generic model of the SXM has a next-state function which moves the machine from the current state to the next state by invoking the corresponding processing function. In

the T-SXM algorithm, this next-state function is considered as a function within the SXM.

Input: *currentState*, *processingFunction*
Output: *nextState*

If *currentState* = null **or** *processingFunction* = null, **then**:
    Return null

nextState = null;

For *transition* **in** *Transitions*:
    If *transition* maps *currentState* **and** *processingFunction* to a valid *newState*, **then**:
        nextState = *transition*'s *newState*
        Stop the loop

Return *nextState*

Figure 6. Algorithm for the next-state function

V. CASE STUDY

A. *The vending machine system*

To show how the T-SXM algorithm works in practice, we consider the implementation of a simple vending machine system written as a standalone Java class. To test the correctness of the implemented vending machine system, we first need to model it as an SXM model with memory, states, transitions, and processing functions. Then, we will use the T-SXM algorithm to transform the vending machine implementation into an SXM specification called X-Vending Machine. After the transformation, the behaviours of the original vending machine system can be tested via the X-Vending Machine.

The vending machine system can be described as follow: the machine is initially at the idle state waiting for a customer. When a customer comes, a product will be selected to start the purchase process. The machine will then confirm the price of the selected product and prompt the customer to insert money. Assuming that the money inserted is a positive integer with unlimited value. Once the amount of money inserted is sufficient, the machine will issue the product and do the calculation to check if change needs issuing. After that, the machine goes back to the idle state. Figure 7 illustrates the implementation of this vending machine system in the Java programming language.

```java
public class VendingMachine {
    private List<Integer> prices;
    private int selectedIndex, amountInserted, change;

    public VendingMachine() {
        prices = new ArrayList<>();
        prices.add(90);
        prices.add(150);
        prices.add(200);

        selectedIndex = -1;
        amountInserted = 0;
        change = 0;
    }

    public void selectProduct(int selectedIndex) {
        if (selectedIndex != -1 && selectedIndex < prices.size()) {
            this.selectedIndex = selectedIndex;
            System.out.println("Product selected has the price of " + prices.get(selectedIndex));
        }
        else {
            System.out.println("Invalid selection.");
        }
    }

    public void insertMoney(int amount) {
        change = amountInserted - prices.get(selectedIndex);
        if (change < 0) {
            amountInserted += amount;
            System.out.println("Amount inserted: " + amountInserted);
        }
        else {
            System.out.println("Sufficient amount received.");
        }
    }

    public void issueProduct() {
        if (amountInserted > 0 && change >= 0) {
            prices.remove(selectedIndex);
            System.out.println("Product issued.");
            if (change > 0) {
                System.out.println("Change issued: " + change);
            }
            else {
                System.out.println("Cannot issue change.");
            }
        }
        else {
            System.out.println("Cannot issue product.");
        }
    }

    public void terminateProcess() {
        selectedIndex = -1;
        amountInserted = 0;
        change = 0;
        System.out.println("Process terminated.");
    }
}
```

Figure 7. Vending machine implementation

With the above description, the vending machine system can be modelled as an SXM with three states:

- **Idle State (initial state):** in which the machine is awaiting a customer.
- **Awaiting Payment State:** in which the machine is awaiting money from the customer.
- **Complete State:** in which the machine issues the product and change (if has) for the customer.

The memory of the X-Vending Machine is a list of four elements which also map correspondingly with the variables used in the implementation showed in Figure 7:

- **prices:** a list of integers representing the prices of the products that the machine has.
- **selectedIndex:** an integer that holds the index of the product the customer has selected. The initial value is -1 which means that no product has been selected.
- **amountInserted:** an integer that keeps track of the amount of money inserted.
- **change:** an integer that is updated every time a new amount of money is inserted.

In the implementation showed in Figure 7, the vending machine has four functions; therefore, the X-Vending Machine also has four functions with identical logics:

- **selectProduct(int index)**
- **insertMoney(int amount)**
- **issueProduct()**
- **terminateProcess()**

Figure 8 shows the state diagram of the X-Vending Machine.

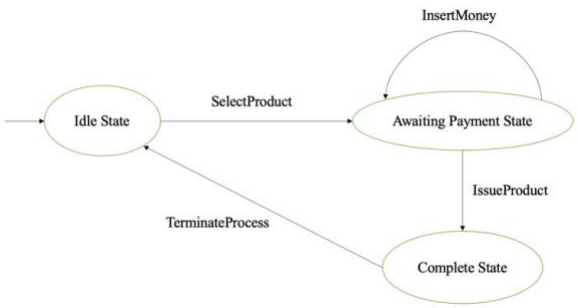

Figure 8. X-Vending Machine state diagram

Based on the above description, specifications, and state diagram, we used the T-SXM algorithm to transform the implementation showed in Figure 7 into an SXM model which is the X-Vending Machine implementation. In the next section, we will use the T-SXM algorithm to specify the input sequence which will be used to test the behaviours of the X-Vending Machine.

B. Testing with the T-SXM algorithm

In this section, we explain how the T-SXM algorithm works by testing the X-Vending Machine with different input values. At the moment, the T-SXM is just an algorithm implemented as a Java class; therefore, it does not have a dedicated user interface and also, the test cases have to be inferred manually by the developers. However, the T-SXM algorithm allows the developers to test the system as a whole by specifying the input values and the transitions in the SXM.

From the implementation in Figure 7, the X-Vending Machine should have three products with the prices of 90, 150, and 200 respectively. Let us consider a scenario when a customer comes and wants to purchase the first product which has the price of 90. The machine is initially at the Idle State. When the customer chooses the product, it will invoke the **selectProduct(int index**) processing function which will update the selectedIndex in the memory to 0 and moves the machine to the Awaiting Payment State. The customer will then insert an amount of money of 100. Thus, while staying in the same state, the machine invokes the **insertMoney(int amount)** processing function with the input value is 100. As the amount of money is sufficient, we need to tell the machine to invoke the **issueProduct()** processing function and move to the Complete State. Once the product and change are issued, we also need to tell the machine execute the **terminateProcess()** processing function to reset its memory and move back to the Idle State.

With the above scenario, the X-Vending Machine should work properly by issuing the product and a change of 10 to the customer. Figure 9 demonstrates the input sequence described in the scenario while Figure 10 shows the output results when the X-Vending Machine executes inputs.

```java
/**
 * Define the input sequence
 *
 * @return
 */
private List<Pair<Object, String>> setupInputSequence() {
    List<Pair<Object, String>> inputSequence = new ArrayList<>();

    // select product at index 0
    inputSequence.add(new Pair<>( a: 0,  b: "selectProduct"));

    // insert 100p
    inputSequence.add(new Pair<>( a: 100,  b: "insertMoney"));

    // issue product
    inputSequence.add(new Pair<>( a: null,  b: "issueProduct"));

    // terminate process
    inputSequence.add(new Pair<>( a: null,  b: "terminateProcess"));

    return inputSequence;
}
```

Figure 9. Input sequence for the described scenario

```
Current state: IdleState
Input value for this state: 0
Processing function for this state: selectProduct
Transition: IdleState --> AwaitingPaymentState
Variable named "selectedIndex" updated.
--> Processing function output: Product selected has the price of 90
Moved to state: AwaitingPaymentState


Current state: AwaitingPaymentState
Input value for this state: 100
Processing function for this state: insertMoney
Transition: AwaitingPaymentState --> AwaitingPaymentState
Variable named "amountInserted" updated.
Variable named "change" updated.
--> Processing function output: Amount inserted: 100
Moved to state: AwaitingPaymentState


Current state: AwaitingPaymentState
Input value for this state: null
Processing function for this state: issueProduct
Transition: AwaitingPaymentState --> CompleteState
Variable named "prices" updated.
--> Processing function output: Product issued.
--> Processing function output: Change issued: 10
Moved to state: CompleteState


Current state: CompleteState
Input value for this state: null
Processing function for this state: terminateProcess
Transition: CompleteState --> IdleState
Variable named "selectedIndex" updated.
Variable named "amountInserted" updated.
Variable named "change" updated.
--> Processing function output: Process completed.
Moved to state: IdleState
```

Figure 10. Outputs from the X-Vending Machine

VI. EVALUATION

The T-SXM algorithm is not a complete tool that can be used in practice. It does not have a user interface and it cannot generate test cases automatically like the other testing tools analysed in section II.B. However, the T-SXM algorithm has demonstrated the potential of using SXM models to test the SUT as a whole, which enables the developers to test the correctness of their implemented systems and thus, resolves the problem of effectiveness discussed in the Introduction section. Also, with the example showed in section V, the T-SXM algorithm has proved the capability of transforming any implemented systems into SXM models which can be used to generate test cases.

By researching and applying the theory of X-Machine and SXM to create the T-SXM algorithm, we realized some limitations that could potentially affect the testing process. As we noted earlier, the X-Vending Machine could not process the inputs and move to the next state automatically. Therefore, we had to explicitly tell the machine what to do with an input value. This can have both advantages and disadvantages. The only advantage is that the developers can test the behaviours of individual functions in the system by telling the machine to do what it should not do. By doing that, the functions have to do their jobs to prevent unexpected inputs and the developers can have a good view of how well the functions are performing. On the contrary, as agile developers, we understand the importance of automatic testing and how other agile developers favour it. Therefore, without the automatic generation of test cases, the T-SXM algorithm would not stand a chance against the other testing tools. The reason why the T-SXM algorithm could not generate test cases automatically based on the inputs is that the current theory of the X-Machine and SXM does not have a mechanism to define the pre-conditions for the machine to process the inputs and determine which processing function it should invoke and which state it should move to.

Furthermore, the T-SXM algorithm cannot interpret an implemented system (e.g., a Java class) and automatically transform it into an SXM model. In the example showed in section V, the X-Vending Machine was manually implemented by the researchers using the T-SXM algorithm. This leaves a big disadvantage for the approach as it will take a significant amount of time for the developers to transform their already implemented system into SXM models. Also, currently, the T-SXM algorithm can only work with standalone Java classes, which means if a system consists of multiple classes, the algorithm can only treat each class as a standalone SXM and test them separately.

VII. FUTURE RESEARCH

With the limitations pointed out in section VI, we will continue researching to improve the T-SXM algorithm. The first aspect we will research is to find out a way to define the pre-conditions in the SXM model so that it can automatically make transitions and invoke processing functions when receiving an input. Also, we will further investigate communicating stream X-Machines which will allow the T-SXM algorithm to model a complex system as multiple SXMs that can communicate with each other. Finally, we will make the algorithm to interpret and transform an implemented system into SXM models automatically so that the developers will not need to do all the hard work themselves.

VIII. CONCLUSION

In general, this paper has presented the T-SXM algorithm which has demonstrated the capability of automated testing by transforming any implemented system into SXM models using their specifications. Although the algorithm has a number of limitations, it has its own novelty in improving the testing process for agile developers. As main researchers in this project, we consider that this T-SXM algorithm is the first successful milestone in a long-term progress. We will do further research to improve the tool and make more effective for testing object-oriented systems, especially the Java programming language.

REFERENCES

[1] Sharma, A., Misra, P.K. (2017) Aspects of enhancing security in software development life cycle. Adv. Comp. Sci. Technol. 10 (2), 203–210.

[2] Aljawarneh, S.A., Alawneh, A., Jaradat, R. (2017) Cloud security engineering: early stages of SDLC. Future Gener. Comp. Syst. 74, 385–392.

[3] Mall, R. (2018) Fundamentals of Software Engineering. PHI Learning Pvt. Ltd..

[4] Fitzgerald, B., Stol, K.J. (2017) Continuous software engineering: a roadmap and agenda. J. Syst. Softw. 123, 176–189.

[5] Pradhan, S., Ray, M. and Swain, S.K. (2019) Transition coverage based test case generation from state chart diagram. *Journal of King Saud University-Computer and Information Sciences*.

[6] Mathur, A.P. (2013) Foundations of Software Testing, 2/e. Pearson Education India.

[7] Bohme, M., Pham, V.T., Roychoudhury, A. (2017) Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Software Eng.

[8] Zhou, Z.Q., Sinaga, A., Susilo, W., Zhao, L., Cai, K.Y. (2018) A cost-effective software testing strategy employing online feedback information. Inf. Sci. 422, 318–335.

[9] Boussaid, I., Siarry, P. and Ahmed-Nacer, M. (2017) A survey on search-based model-driven engineering. *Automated Software Engineering*, *24*(2), pp.233-294.

[10] Mostowski, W. (2019) Model-based fault injection for testing gray-box systems. J. Logical Algebraic Methods Programm. 103, 31–45.

[11] Da Silva, A.R. (2015) Model-driven engineering: a survey supported by the unified conceptual model. Comp. Languages, Syst. Struct. 43, 139–155.

[12] Pradhan, S., Ray, M., Patnaik, S. (2019) Coverage criteria for state-based testing: a systematic review. Int. J. Inf. Technol. Project Manage. (IJITPM) 10 (1), 1–20.

[13] Utting, M., Legeard, B., Bouquet, F., Fourneret, E., Peureux, F., Vernotte, A. (2016) Recent advances in model-based testing. Advances in Computers 101, 53–120.

[14] Bonifacio, A.L., Moura, A.V., Simao, A.S. (2008) A generalized model-based test generation method. *6th IEEE International Conferences on Software Engineering and Formal Methods*, Cape Town, South Africa; 139–148.

[15] Dorofeeva, R., El-Fakih, K., Yevtushenko, N. (2005) An improved conformance testing method. *Formal Techniques for Networked and Distributed Systems*, Taipei, Taiwan; 204–218.

[16] Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R., Yevtushenko, N. (2010) FSM-based conformance testing methods: a survey annotated

with experimental evaluation. *Information and Software Technology*; **52**(12):1286–1297.

[17] Simao AS, Petrenko A, Yevtushenko N. (2009) Generating reduced tests for FSMs with extra states. In *Testing of Software and Communication Systems*, Vol. 5826. Springer: Berlin / Heidelberg; 129–145.

[18] Chaves Pedrosa, L.L. and Vieira Moura, A., (2013). Incremental testing of finite state machines. *Software Testing, Verification and Reliability*, *23*(8), pp.585-612.

[19] Kumar, S., Rathore, S.S. and SpringerLink (Online service) (2018) *Software Fault Prediction: A Road Map*, Springer Singapore, Singapore.

[20] Catal, C. and Diri, B. (2009) Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, *179*(8), pp.1040-1058.

[21] Kalsoom, A., Maqsood, M., Ghazanfar, M.A., Aadil, F. and Rho, S. (2018) A dimensionality reduction-based efficient software fault prediction using Fisher linear discriminant analysis (FLDA). *The Journal of Supercomputing*, *74*(9), pp.4568-4602.

[22] Nasrabadi, M.Z., Parsa, S. and Kalaee, A. (2018) Format-aware Learn&Fuzz: Deep Test Data Generation for Efficient Fuzzing. *arXiv preprint arXiv:1812.09961*.

[23] Ipate, F., Gheorghe, M. and Holcombe, M. (2003) Testing (Stream) X-machines. *Applicable Algebra in Engineering, Communication and Computing*, *14*(3), pp.217-237.

[24] Chow, T. S. (1978) Testing Software Design Modelled by Finite State Machines. *IEEE Transactions on Software Engineering*, 4(3), 178-187.

[25] Fujiwara, S., Bochmann, G.v., Khendek, F., Amalou, M., Ghedamsi, A. (1990) Test selection based on finite state models. Publication #716, Departement d'informatique et de recherche operationnelle, University of Montreal.

[26] Fujiwara, S. and Bochmann, G.v. (1991) Testing non-deterministic finite state machines. Publication #758, Departement d'informatique et de recherche operationnelle, University of Montreal.

[27] Holcombe, M. and Ipate, F. (1998) Correct Systems: Building a Business Process Solution. Berlin: Springer.

[28] Eilenberg, S. (1974) Automata, languages and machines. Vol. A, Academic Press.

[29] Ipate, F. and Holcombe, M. (1998) A method for refining and testing generalized machine specifications. Intern. J. Computer Math. **68**, 197–219.

[30] Ipate, F. and Holcombe, M. (1998) Specification and testing using generalized machines: a presentation and a case study. J. Software Testing, Verification and Reliability **8**, 61–81.

[31] Ipate, F. and Gheorghe, M. (2009) Testing non-deterministic stream X-machine models and P systems. *Electronic Notes in Theoretical Computer Science*, *227*, pp.113-126.

[32] Chin, L.S., Worth, D.J. and Greenough, C. (2007) A survey of software testing tools for computational science. *Software Engineering Group Computational Science & Engineering Department*.

[33] Louridas, P. (2005) Junit: Unit Testing and Coding in Tandem. *IEEE Software*. 22 (4), pp. 12-15. Merayo, M.G., Núñez, M. and Hierons, R.M. (2009) Testing timed systems modelled by stream x- machines. *Software & Systems Modeling*. 10(2), pp. 201–217.

[34] Smeets, N, Simons, A.J.H (2009) Comparing the Effectiveness of Automatically Generated Tests by Randoop, JWalk and μJava with jUnit Tests. Research report, University of Sheffield/University of Antwerp.

[35] Simons, A.J.H., Bogdanov, K. and Holcombe, M. (2014) *Complete Functional Testing using Object Machines*. [online]. Report Number: CS -01 -18. University of Sheffield. Available from: http://www.dcs.shef.ac.uk/intranet/research/public/resmes/CS0204.pdf [Accessed 12 June 2020].

[36] Schneider, A. (2000) *Junit best practices*. Available from: http://www.javaworld.com/article/2076265/testing-debugging/junit-best-practices.html [Accessed 12 June 2020].

[37] Csallner, C. and Smaragdakis, Y. (2004) JCrasher: An automatic robustness tester for java. Software - Practice and Experience. 34 (11), pp. 1025-1050.

[38] Ipate, F. (2003) "On the Minimality of Stream X-machines", *The Computer Journal*, vol. 46, no. 3, pp. 295-306.