UML State Machine Diagram Driven Runtime Verification of Java Programs for Message Interaction Consistency

Xuandong Li, Xiaokang Qiu, Linzhang Wang, Bin Lei


Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.
UML State Machine Diagram Driven Runtime Verification of Java Programs for Message Interaction Consistency

Xuandong Li, Xiaokang Qiu, Linzhang Wang, Bin Lei
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, P.R.China 210093
Department of Computer Science and Technology, Nanjing University, Nanjing, P.R.China 210093
lxd@nju.edu.cn

W. Eric Wong
Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA
ewong@utdallas.edu

ABSTRACT
In object-oriented programs, we often need to set some restrictions on the temporal orders of the message receiving for objects, which forms a class of safety requirements. In this paper, we use UML state machine diagrams to design specifications, and present an approach to runtime verification of Java programs, which is focused on the temporal order of message receiving based consistency verification between the behavior of state machine diagrams and the program execution traces. In the approach, we first instrument the program under verification so as to gather the program execution traces related to a given state machine diagram. Then we drive the instrumented program by random test cases so as to generate the program execution traces. Finally we check if the collected program execution traces are consistent with the behavior of the state machine diagram. Our approach can be used to detect not only the program bugs resulting from the wrong temporal orders of message receiving, but also the imperfect state machine models constructed in reverse engineering for legacy systems, and leads to a testing tool which may proceed in a fully automatic fashion.

Key words: Runtime verification, Java, UML state machine diagrams.

1. INTRODUCTION
In object-oriented programs, the objects receive the messages, execute their corresponding methods, and change their states. The temporal orders of the message receiving form an important aspect of the object behavior. We often need to set some restrictions on the temporal orders of the message receiving for objects, which forms a class of safety requirements.

The Unified Modeling Language (UML) is a general purpose visual modelling language that is designed to specify, visualize, construct and document the artifacts of software systems [1,2]. Since UML became a standard in OMG in 1997, UML models have become a main class of artifacts in software development processes. UML provides a number of diagrams to describe particular aspects of software artifacts. These diagrams can be classified depending on whether they are intended to describe structural or behavioral aspects of systems. UML state machine diagrams describe behavioral aspects of systems, and are used in design processes to model the dynamic behavior of objects, emphasizing the flow of control from state to state where event communication and data determine possible sequences of states. The temporal orders of the message receiving, which should be confirmed by during an object’s lifecycle, can be reflected in the UML state machine diagram which describes the behavior of the object.

Runtime verification [3] is a lightweight approach to program reliability. Its basic idea is to gather information during program execution and use it to conclude properties about the program, either during testing or in operation. Given a trace of program execution, we report success if the trace satisfies the program specification, and failure if a fault is detected [4].

In this paper, we use UML state machine diagrams as design specifications, and present an approach to runtime verification of Java programs, which is focused on the temporal order of message receiving based-consistency verification between the behavior of state machine diagrams and the program execution traces. In the approach, we first instrument the program under verification so as to gather the program execution traces related to a given state machine diagram. Then we drive the instrumented program by random test cases so as to generate the program execution traces. Finally we check if the collected program execution traces are consistent with the behavior of the state machine diagram, which means that the temporal orders of the message receiving occurring in the program traces are consistent with the ones occurring in the state machine diagram. The verification process is depicted in Figure 1.

The paper is organized as follows. In the next section, we
an automated teller machine (ATM) simulation system. For example, Figure 2 gives a simple state machine diagram in which a state machine is a collection of vertices and arcs \[1,2\]. For a state machine, emphasizing the flow of control from state to state. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An event is an occurrence of a stimulus that can trigger a state transition. A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. Graphically, a state machine diagram is a collection of vertices and arcs \[1,2\]. For example, Figure 2 gives a simple state machine diagram in an automated teller machine (ATM) simulation system in [5].

2. UML STATE MACHINE DIAGRAMS

UML state machine diagrams are used to model the dynamic aspects of a system. A state machine diagram shows a state machine, emphasizing the flow of control from state to state. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An event is an occurrence of a stimulus that can trigger a state transition. A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. Graphically, a state machine diagram is a collection of vertices and arcs \[1,2\]. For example, Figure 2 gives a simple state machine diagram in an automated teller machine (ATM) simulation system in [5].

In this paper, we focus our attention on the temporal orders of the message receiving, which are reflected in the state machine diagrams and should be conformed to during an object’s lifecycle. For a state machine diagram, an event labelled on a transition usually corresponds a message receiving so that its behavior may correspond to the message receiving flows. For example, according to the state machine diagram in Figure 2, one scenario is that the ATM object starts from the initial state, is ready for the customers by receiving the message switchOn(), serves a customer by receiving the message cardInserted(), returns IDLE state by receiving the message ejectCard(), and performs a shutdown by receiving the message switchOff(), which forms a normal sequence of message receiving. But it is illegal that during ATM serving a customer by receiving the message cardInserted(), a message switchOff() arrives before receiving the ejectCard(), which results in the customer’s operation aborting, and the information is lost.

In this paper, since we are focused only on the temporal order of message receiving-based consistency verification between the behavior of state machine diagrams and the program execution traces, we just need to use a simplified version of state machine diagrams \[1,2\] in which a state machine diagram just consists of states, transitions, and events labelled on transitions. We do not consider the notations of activities, actions, hierarchical state configurations, and of composite states in state machine diagrams. It is not difficult to transform a state machine diagram with the above complex notations to the simple version we consider. For solving the verification problem, we formalize state machine diagrams as follows.

**Definition 1.** A state machine diagram is a tuple \(G = (S, E, T, s^0, c)\) where \(S\) is a finite set of states; \(E\) is a finite set of events; \(T\) is a transition relation whose elements are of the form \((s, e, s')\) where \(s, s' \in S\) and \(e \in E\) is an event; \(s^0 \in S\) is an initial state; \(c\) is the class whose instance behavior is described by \(G\).

We use state sequences to represent the behavior of state machine diagrams. Any state sequence is of the form

\[
\sigma = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_{m-1}} s_m \xrightarrow{e_m} s_{m+1}
\]

where \(s_i (0 \leq i \leq m+1)\) is a state and \(e_i (0 \leq i \leq m)\) is an event, which represents a behavior of a state machine diagram such that the system starts at the state \(s_0\), then change to \(s_1\) through the transition activated by the event \(e_0\), and so on.

**Definition 2.** Let \(G = (S, E, T, s^0, c)\) be a state machine
diagram. A state sequence

\[ s_0 \xrightarrow{c_0} s_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{m-1}} s_m \xrightarrow{c_m} s_{m+1} \]

is a behavior of G if and only if

\[ s_0 = s^0, \text{ and } (s_i, e_i, s_{i+1}) \in T \text{ for each } i \ (0 \leq i \leq m). \]

\[ \square \]

3. RUNTIME VERIFICATION OF JAVA PROGRAMS

Now we consider runtime verification of Java programs for UML state machine diagrams. The verification process consists of three main steps: program instrumenting, program executing driven by random test cases, and consistency checking, which is depicted in Figure 1.

For a program under verification, we gather its execution traces by running its instrumented version. The test cases used to drive the programs are generated randomly. The first reason for selecting the random method is its inexpensive cost. Secondly, it is automatic and can be applied in almost any system, which is just what we want actually, an important problem with random testing is how many test cases should be created. Since we are automatically generating our test cases, this problem is handled by allowing the tool to run until there is sufficient confidence in the result depending on the application. That is, we think the test cases are sufficient when the tool has been running for a duration long enough, or when an apparent and believable result can be concluded, i.e. an inconsistent case is detected.

3.1 Program Instrumentation

The problem we approach is to check Java programs for the specifications expressed by state machine diagrams, which focus on the temporal orders of message receiving. Thus, for a Java program under verification, we need to insert some statements into its source code for gathering the program execution traces, and the program execution traces we gather are a sequence of events corresponding to receiving messages.

In a Java program, the first statement execution in a method is corresponding to an event for a message receiving. Thus we insert the statements for gathering the information in the beginning of each related method definition. Given a state machine diagram \( G = (S, E, T, s^0, c) \) where \( E_M \subseteq E \) is the set of the events corresponding to message receiving, when an event in \( E_M \) for a message receiving happens, the information we need to log includes the message, its receiver, and the class which the receiver belongs to. The instrumentation algorithm depicted in Figure 3 runs as follows. First we scan the program to parse the source code into a file of tokens. Then we check each token for recognizing the related method definitions. If a definition of method \( m \) corresponding to an event in \( E_M \) is found, then we insert the code segment \( \text{Log\_Receiving\_Event} \) depicted in Figure 3 before the first statement in the method definition, which is used to gather the information about the message and its receiver.

A message receiver we log is represented by the character string consisting of its class name and a hash code, which is provided by Java API class method \( \text{Object}\rightarrow\text{toString} \). The \( \text{toString} \) method defined by class \( \text{Object} \) does return distinct hash codes for distinct objects, but since this is implemented

```
try{ synchronized(this) {
    StackTraceElement[] ste =
        new Throwable().getStackTrace();
    java.io.RandomAccessFile receiveLog =
        new java.io.RandomAccessFile(log, "rw");
    receiveLog.seek(receiveLog.length());
    receiveLog.writeBytes("" + ste[0].getMethodName() +
                        ""," + ste[0].getClassName() + "");
    receiveLog.close();
}
```

...}

Figure 3: Instrumentation Algorithm and Inserted Code Segment

by converting the internal address of the object into a hash code, it is still possible for different objects that exist in different time to return the same hash code. To determine the life cycles of these dynamic objects, we need to instrument the finalizer of the concerned classes. Whenever one object is finalized, its hash code is logged so that we know what a hash code exactly refers to at different times.

3.2 Consistency Checking

According to the algorithm for instrumenting programs given in Section 3.1, by running an instrumented program we can get the program execution traces related to a given state machine diagram. In the following, we give the solution to checking the consistency between the program execution traces and the behavior of the state machine diagram.

From the algorithm for instrumenting programs, it follows that a program execution trace we gather can be represented by a sequence of the form

\[ (m_0, r_0, c_0) \rightarrow (m_1, r_1, c_1) \rightarrow \cdots \rightarrow (m_n, r_n, c_n) \]

where for any \( i \ (0 \leq i \leq n) \), \( m_i \) is a message, \( r_i \) is the receiver of \( m_i \) represented by a hash code, and \( c_i \) is the class which \( r_i \) belongs to.

Let \( \sigma = (m_0, r_0, c_0) \rightarrow (m_1, r_1, c_1) \rightarrow \cdots \rightarrow (m_n, r_n, c_n) \) be a program execution trace, and \( G = (S, E, T, s^0, c) \) be a state machine diagram. Given a receiver \( r_k \ (0 \leq k \leq n) \) whose class is \( c \), by removing all \( (m_i, r_i, c_i) \ (0 \leq i \leq n) \) such that \( r_i \neq r_k \vee c_i \neq c \) from \( \sigma \) we get a sequence \( \sigma' \) which is called the projection of \( \sigma \) on \( G \).
Let $G = (S, E, T, s^0, c)$ be a state machine diagram where $E_M \subseteq E$ is the set of the events corresponding to message receiving. For a behavior $\rho$ of $G$ the form

$$\rho = s_0 \stackrel{e_0}{\rightarrow} s_1 \stackrel{e_1}{\rightarrow} \cdots \stackrel{e_{m-1}}{\rightarrow} s_m \stackrel{e_m}{\rightarrow} s_{m+1},$$

we can get an event sequence $\tau = e_0' e_1' \cdots e_m'$. By removing any $e_i \notin E_M (0 \leq i \leq m)$ from $\tau$, we get an event sequence $\tau'$ which is called the message trail of $\rho$.

**Definition 3.** Let $G = (S, E, T, s^0, c)$ be state machine diagram, and $\sigma$ be a corresponding program execution trace. $\sigma$ is consistent with the behavior of $G$ if for each projection of $\sigma$ on $G$ of the form $(e_0, r, c) \cdot (m_1, r, c) \cdot \cdots \cdot (m_n, r, c)$, there is a behavior of $G$ whose message trail of the form $e_0 \cdot e_1 \cdot \cdots \cdot e_n$ is such that $e_i$ is corresponding to the receiving of $m_i$ for any $i (0 \leq i \leq n)$.

According to the above definition, for solving the consistency checking problem, we need to check if there is a behavior of a given state machine diagram satisfying the condition given in the definition. We know that for a state machine diagram, there could be infinite behaviors and the number of its behavior could be infinite. In the following, we show how to solve the consistency checking problem based on the investigation of a finite set of finite behavior of the state machine diagram.

Let $G = (S, E, T, s^0, c)$ be a state machine diagram where $E_M \subseteq E$ is the set of the events corresponding to message receiving, and $\sigma$ be a corresponding program execution trace. For any projection $\sigma'$ of $\sigma$ on $G$ of the form

$$(m_0, r, c) \cdot (m_1, r, c) \cdot \cdots \cdot (m_n, r, c),$$

a behavior $\rho$ of $G$ of the form

$$s_0 \stackrel{e_0}{\rightarrow} s_1 \stackrel{e_1}{\rightarrow} \cdots \stackrel{e_{m-1}}{\rightarrow} s_m \stackrel{e_m}{\rightarrow} s_{m+1}$$

is the reference of $\sigma'$ if the following condition holds:

- the message trail of $\rho$ of the form $e_0' \cdot e_1' \cdot \cdots \cdot e_n'$ is such that $e_i'$ is corresponding to the receiving of $m_i$ for any $i (0 \leq i \leq n)$, and
- for any $s_i$ and $s_j$ ($0 \leq i < j \leq m$), if there is not any $e_k$ ($i \leq k \leq j$) such that $e_k \in E_M$ then $s_i \neq s_j$.

Notice that if $\sigma$ is finite, then the reference of any projection of $\sigma$ on $G$ is finite.

**Theorem 1.** Let $G = (S, E, T, s^0, c)$ be a state machine diagram, and $\sigma$ be a corresponding program execution trace. $\sigma$ is consistent with the behavior of $G$ if and only if for any projection $\sigma'$ of $\sigma$ on $G$, there is a behavior of $G$ which is the reference of $\sigma'$.

**Proof.** It is clear that one half of the claim holds: if for any projection $\sigma'$ of $\sigma$ on $G$, there is a behavior of $G$ which is the reference of $\sigma'$, then $\sigma$ is consistent with the behavior of $G$. The other half of the claim can be proved as follows.

Suppose that $\sigma$ is consistent with the behavior of $G$, and $\sigma' = (m_0, r, c) \cdot (m_1, r, c) \cdot \cdots \cdot (m_n, r, c)$ is a projection of $\sigma$. From Definition 3 it follows that there is a behavior $\rho$ of $G$ of the form

$$s_0 \stackrel{e_0}{\rightarrow} s_1 \stackrel{e_1}{\rightarrow} \cdots \stackrel{e_{m-1}}{\rightarrow} s_m \stackrel{e_m}{\rightarrow} s_{m+1}$$

such that its message trail of the form $e_0' \cdot e_1' \cdot \cdots \cdot e_n'$ satisfies that $e_i'$ is corresponding to the receiving of $m_i$ for any $i (0 \leq i \leq n)$. If $\rho$ is not the reference of $\sigma'$, then there are $s_i$ and $s_j$ ($0 \leq i < j \leq m$) such that $s_i = s_j$ and that is not any $e_k$ ($i \leq k \leq j$) such that $e_k \in E_M$. In this case, by removing the subsequence $s_i \stackrel{e_i}{\rightarrow} s_{i+1} \stackrel{e_{i+1}}{\rightarrow} \cdots \stackrel{e_{j-1}}{\rightarrow} s_j$, if the following condition holds:

$$s_i \stackrel{e_i}{\rightarrow} s_{i+1} \stackrel{e_{i+1}}{\rightarrow} \cdots \stackrel{e_{j-1}}{\rightarrow} s_j$$

from $\rho$, we get a behavior $\rho'$ of $G$ such that $\rho$ and $\rho'$ have the same message trail. Thus, by repeating the above process, we can get a behavior of $G$ which is the reference of $\sigma'$, which implies the claim holds.

Based on the above theorem, we can develop an algorithm (which is depicted in Figure 4) to check if all traces in a trace file, which is generated by running an instrumented program, are consistent with the behavior of a given state machine diagram $G = (S, E, T, s^0, c)$. For a projection $\sigma$ of a program execution trace on $G$, a behavior $\rho$ of $G$ is a prefix for checking if it may be extended into a reference of $\sigma$, i.e., there could be a state sequence $\rho'$ such that $\rho \cdot \rho'$ is the reference of $\sigma$. For each trace in the trace file, for each projection of the trace, the algorithm traverses the state space of $G$ in a depth first manner starting from the initial state. The path in the state space that we have so far traversed is stored in the variable $currentpath$. For each new node that we discover, we first check whether it is such that the behavior corresponding to $currentpath$ is the reference of the projection. If yes, then it means that the projection is consistent with the behavior of $G$. Then we check if the new node that we discover is such that the behavior corresponding to $currentpath$ is a prefix for checking. If yes, then we add the new node to the current path and start the search for the next reference.
from it, otherwise we backtrack. If no reference of the projection is discovered during the search, then it means that the trace is not consistent with the behavior of G. The complexity of the algorithm is proportional to the number of the prefixes for checking and to the size of the longest trace in the trace file.

3.3 Support Tool and Case Study

With the work presented in this paper, we aim to develop an automatic support tool for testing, which may help us to reduce the testing cost. This tool can help us to detect the inconsistency between the behavior implemented by the program and the expected behavior specified by the state machine diagrams. The tool may proceed in a fully automatic fashion, and we can drive this tool after we leave our office in the evening, and see the results the next morning.

Based on the work presented in this paper, we have implemented a prototype of this kind of tool in Java. The tool accepts a Java program under verification, instruments the program according to the given state machine diagrams, drives the instrumented program to execute on a set of random test cases, and reports the errors which result from the inconsistency with the behavior of the state machine diagrams.

With the tool, we have conducted several case studies for showing the potential and usability of the approach presented in this paper. One case study is based on a microwave oven simulation system which is implemented in Java with 17 classes and 113 methods. We use the state machine diagram of the microwave oven class as the design specification, which is depicted in Figure 5. There are eleven states and sixty-one transitions in the specification. In addition to the bugs embedded manually, by running the tool an inconsistent case is detected, which results from a program bug. This inconsistent case is that the program execution traces show that at the state Cooking, after the event openDoor is triggered, the event pause follows, but according to the state machine diagram, pause is not supposed to occur after openDoor. By looking into the source code, we find a redundant method invocation, which is a program bug. This bug has not been found by the previous testing work since it does not produce a wrong external output.

Another case study is based on an official retirement insurance system (ORIS) which is implemented in Java with 17 classes and 241 methods. ORIS is a real project in the industry, and was already deployed in February 2005. Because the specification of ORIS is not described in UML, we need to construct the state machine diagrams for this case study according to the design specification of ORIS. We select the person class to construct its state machine diagram in which there are nine states and seventeen transitions, which is depicted in Fig 6. In the beginning, the tool reports several inconsistent cases, but by checking the original specification we discover that all those inconsistent cases result from an imperfect state machine diagram. After modifying the state machine diagram, no inconsistent case is detected by the tool. This case study also indicates that our approach can be used to detect the imperfect state machine models constructed in reverse engineering for legacy systems.

4. CONCLUSION

In this paper, we give an approach to runtime verification of Java programs for UML state machine diagrams, which is focused on the temporal order of message receiving based consistency verification between the behavior of state machine diagrams and the program execution traces. The presented approach can be used to detect not only the program bugs resulting from the wrong temporal orders of message receiving, but also the imperfect state machine models constructed in reverse engineering for legacy systems, and leads to a testing tool which may proceed in a fully automatic fashion.

There are a number of works on UML state machine diagram based testing [15-18] whose intentions are different from the one of our work, which are focused on deriving test cases and constraints from state machine diagrams. There are also several works [10,11] on verifying Java programs based on model checking techniques whose capacity is restricted by the huge program state spaces. Several works have been focused on synthesis of the temporal order of
message interaction in systems [12-14]. An approach has been presented in [13] for synthesis of the temporal order of message receiving for Java classes.

The runtime verification techniques have been used to detect the concurrency errors such as deadlocks and data races for Java programs [6-9]. In those works, most of the specification languages are based on temporal logic. Although such languages are designed to describe the relations among states and events, they do not give a full support to describe the complete behavior of objects. Compared to those specification languages, UML state machine diagrams are much more expressive for describing the behavior of objects. Furthermore, UML state machine diagrams can come directly from the artifacts generated in software development processes. We know that it is not easy to use formal verification techniques directly in industry because the specification languages in the verification tools are too formal and theoretical to master easily. In industry, it is much more acceptable to adopt UML state machine diagrams as a specification language instead of the temporal logic based languages in formal verification tools.

In addition, since the specifications are described by UML state machine diagrams, our work can also be used to detect the imperfect UML state machine models constructed in reverse engineering for legacy systems, which is another advantage. A similar approach has also been used for runtime verification of Java programs for scenario-based specifications modelled by UML sequence diagrams [19].

In this paper, our work focuses on the runtime verification of Java programs, but the underlying approach and ideas are more general and may also be applied to the runtime verification of the programs in other object-oriented languages.

Acknowledgements
We would like to thank Mr. Jacob Orshalick at the University of Texas at Dallas for implementing the microwave oven simulation system used in our case study. The authors at Nanjing University are supported by the National 863 High-Tech Programme of China (No.2007AA010302), the National Natural Science Foundation of China (No.60425204 and No.60603036), and by the Jiangsu Province Research Foundation (BK2007714).

5. REFERENCES