Test Input Generation using UML Sequence and State Machines Models

Aritra Bandyopadhyay, Sudipto Ghosh
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
{baritra,ghosh}@cs.colostate.edu

Abstract

We propose a novel testing approach that combines information from UML sequence models and state machine models. Current approaches that rely solely on sequence models do not consider the effects of the message path under test on the states of the participating objects.

Dinh-Trong et al. proposed an approach to test input generation using information from class and sequence models. We extend their Variable Assignment Graph (VAG) based approach to include information from state machine models. The extended VAG (EVAG) produces multiple execution paths representing the effects of the messages on the states of their target objects.

We performed mutation analysis on the implementation of a video store system to demonstrate that our test inputs are more effective than those that cover only sequence diagram paths.

Keywords: *class models, model-based testing, sequence models, state machine models, test input generation*

1. Introduction

Current research has focused on using UML models to support system testing. Testing approaches based on UML sequence (or collaboration) diagrams [1, 3, 6] select a set of message paths according to some structural coverage criterion (e.g., all-message-path coverage). Test inputs are generated so that each message path is executed. However, such approaches ignore the effect of a message sequence on the states of the objects participating in the interaction.

Testing approaches based on state machine models [5, 13] select a set of transition sequences using state machine coverage criteria (e.g., coverage of all states, transitions, or transition pairs) and generate test inputs for each of those transition sequences. For example, Offutt et al. [13] state that "a complete sequence is a sequence of state transitions that form a complete practical use of the system. In most realistic applications, the number of possible sequences is too large to choose all complete sequences." Therefore, they suggest that test engineers define meaningful sequences from the diagram. One way to do that is to use the experience and knowledge of the test engineer. Other than that, none of the state machine based testing approaches provide any guidance for selecting meaningful sequences.

The motivation of our work stems from the two problems described above. We assume that the developers have the following design models: (1) a UML class model to specify the structure, (2) state machine models to specify states and transitions for each object as defined in the class diagram, and (3) several sequence models to show critical scenarios of system use. We improve sequence model based testing by considering the effect of each message on the states of the target object. We use sequence model interactions to select key transitions and states from a state machine.

In our approach we combine the information from sequence and state machine models into one testable model. We extend the work of Dinh-Trong et al. [6], which proposed the use of a Variable Assignment Graph (VAG) to combine information from a class and a sequence diagram. We extend a VAG with information from the state machines (if available) for each class defined in the class diagram. We do not necessarily need state machine models of each class, but the more we can use, the more effective is the approach. We performed a mutation analysis of a video store system to demonstrate that our test inputs are more effective than those that cover only sequence diagram paths.

Section 2 summarizes the original VAG generation approach described in Dinh-Trong et al. [6]. Section 3 describes how the EVAG is constructed. Section 4 presents the results of our pilot study. Section 5 describes related work. Section 6 presents our conclusions and outlines directions for future work.

2. Background

The test generation approach of Dinh-Trong et al. [6] uses two types of UML models: class and sequence.

Class model information is needed to create a configuration of objects on which the test is to be performed, such that the configuration conforms to the class diagram. This information includes: (1) constraints on attributes and the class invariant (available in the OCL specification associated with the class diagram), and (2) associations and multiplicities.

Sequence model information in the form of a sequence of operations is needed to ensure that a certain path gets executed. This, in turn, requires that we have (1) the precondition of the operations (available in the OCL specification associated with the class diagram), (2) conditions along the path that must be satisfied (available in the opt, alt, and loop fragments (`CombinedFragments`) of a sequence diagram), and (3) information about where and how the values are defined (available in the sequence diagram through actual parameters that are assigned to formal parameters or variables that hold the return values, and in the class diagram through post-conditions of operations).

First, the information in a class diagram and a sequence diagram is combined to generate a graph structure called Variable Assignment Graph (VAG). Paths are selected from the VAG based on some VAG-based structural coverage criterion (e.g., cover all nodes, cover all edges, or cover all paths). For each path, the constraints along the path are combined into a constraint system. The constraint variables are treated as symbolic variables. Since the authors are not aware of any constraint solver that can solve this type of constraints, they convert the constraints to Alloy [8]. If the Alloy solver finds a solution, the test input is determined from the solution. Otherwise, a new path is selected and the process of constraint generation and solving is repeated.

A VAG is a control flow graph annotated with data-flow information. The graph contains a message node for every sequence diagram message. Edges between message nodes represent the control flow between the corresponding messages. There are also control nodes, which show alternate execution paths of messages from the `CombinedFragments`.

A message node in a VAG may contain any of the following three sections:

- *Condition:* It represents the constraints that must be satisfied in order to send the message. The constraints include the operation pre-condition, a predicate for the existence of the target object of the message, and a predicate for the existence of the link along which the message is to be sent.
- *Control Action:* It represents the implications of sending a message and includes the assignment of the arguments to the formal parameters of the operation invoked by the message.
- *Effect:* It represents the changes in the system after the operation invoked as a result of a message finishes execution and returns. The effect of a message, m, is associated with the message node of the return message corresponding to m. Effects include the post-conditions of the operation invoked by the message.

We use an example extended from Briand et al. [5] to illustrate the steps involved in the generation of a VAG. The example is the UML specification of a video store management system. In Section 3, we use the example to illustrate the generation of an EVAG from a VAG.

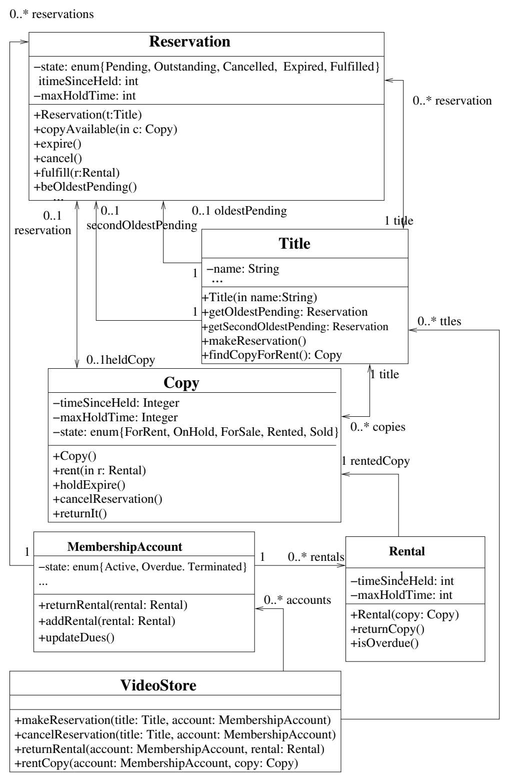


Figure 1. Video Store Class Diagram Extended from Briand et al. [3]

Figure 1 shows a class diagram containing classes, *Copy*, *Title*, *Reservation*, *MembershipAccount*, *Rental*, and *VideoStore*. Figure 2 shows an interaction that occurs when a rental is returned by a member.

Figure 3 shows the VAG generated using the sequence diagram in Figure 2 The nodes are labeled with the operations of the messages they represent. Nodes `a` and `b` are control nodes. Node `a` is used to select one of the two conditional message paths resulting from the `opt` fragment in

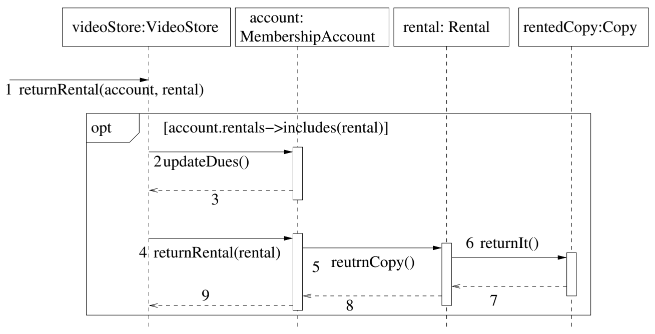


Figure 2. Sequence Diagram Showing Returning a Rental by a Member

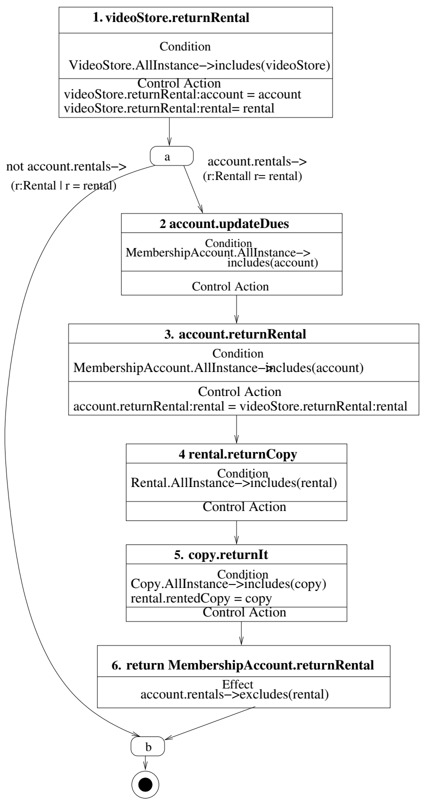

1. videoStore.returnRental
Condition
VideoStore.AllInstance−>includes(videoStore)
Control Action
videoStore.returnRental:account = account
videoStore.returnRental:rental= rental
a
not account.rentals−> (r:Rental | r = rental)
account.rentals−> (r:Rental| r= rental)
2 account.updateDues
Condition
MembershipAccount.AllInstance−> includes(account)
Control Action
3. account.returnRental
Condition
MembershipAccount.AllInstance−>includes(account)
Control Action
account.returnRental:rental = videoStore.returnRental:rental
4 rental.returnCopy
Condition
Rental.AllInstance−>includes(rental)
Control Action
5. copy.returnIt
Condition
Copy.AllInstance−>includes(copy)
rental.rentedCopy = copy
Control Action
6. return MembershipAccount.returnRental
Effect
account.rentals−>excludes(rental)
b

Figure 3. VAG Generated From Figs 1 and 2

the sequence diagram. Each branch is annotated with the corresponding condition for execution of that branch. Node `b` is used to merge them.

Consider message node 3 in Figure 2. It represents message 4 in the sequence diagram. The `Condition` section represents the condition that must exist in order to send the message, `returnRental()` to the instance *account* of the *MembershipAccount* class. The OCL statement in the condition specifies that the recipient instance, `account`, must exist before the message is sent.

The `Control Action` in the node states that the argument is assigned to the formal parameter of the operation, `returnRental()`. The name of the formal parameter is obtained from the signature of the operation, `returnRental()`, in the class diagram. The name of the argument is obtained from the sequence diagram message. To avoid conflict of variable names, all variables are expressed in a fully qualified form. The formal parameter and the argument of `returnRental()` for message 4 are both `rental`. The formal parameter is expressed as `account.returnRental:rental`. The argument, being a local variable of the operation, `returnRental()` (message 1), is expressed as `videoStore.returnRental:rental`.

Node 6 in Figure 2 represents the return message corresponding to the call, `returnRental()`. The `Effects` section of the node records the post-condition of `returnRental()`, which states that the *Rental* instance to be returned is removed from the list of rentals in the account.

Each path from the start node to the end node of the VAG represents a complete execution path in the sequence diagram. The conjunction of the constraints in the edges and `Condition`, `Control Action`, and `Effect` sections along a VAG path together with the class diagram invariant represents the condition that must be fulfilled in order to execute the message path.

3. Proposed Approach

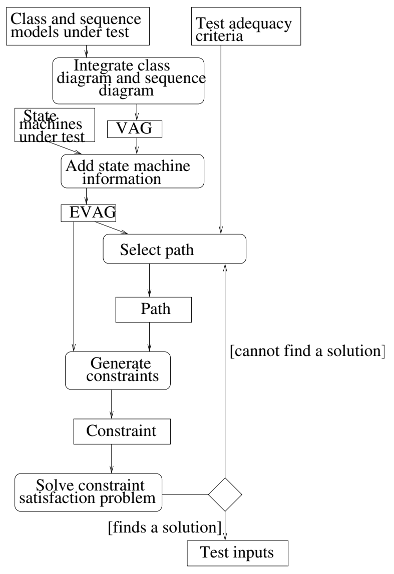

Class and sequence models under test
Test adequacy criteria
Integrate class diagram and sequence diagram
State machines under test
VAG
Add state machine information
EVAG
Select path
Path
[cannot find a solution]
Generate constraints
Constraint
Solve constraint satisfaction problem
[finds a solution]
Test inputs

Figure 4. Our Approach

Figure 4 describes the activities in our approach. We first construct the VAG as before, using a class diagram and a sequence diagram. We augment the VAG with information derived from the state machines of the participating objects to build a testable model called the Extended Variable Assignment Graph (EVAG).

The rest of the test generation process proceeds in the same way as described in Section 2. We use structural coverage criteria to select paths from the EVAG. For any selected path, we generate the path constraints, which along with the class diagram are transformed into an Alloy script. The resulting Alloy script is executed to solve the constraints and the solution is converted to JUnit test cases. In this paper, we only focus on the construction of the EVAG from a VAG and the state machine diagrams.

We assume that each diagram is syntactically correct and also syntactically consistent with each other; most UML drawing tools can perform these checks. Each sequence diagram lifeline has a corresponding class in the class diagram. Each message in the sequence diagram and each call-event and action in the state machines conforms to the corresponding operation signature in the class diagram. The names of class attributes and formal parameters of operations are consistent across the diagrams. Currently, we support only call-events in the state machines and synchronous operation calls in the sequence diagrams.

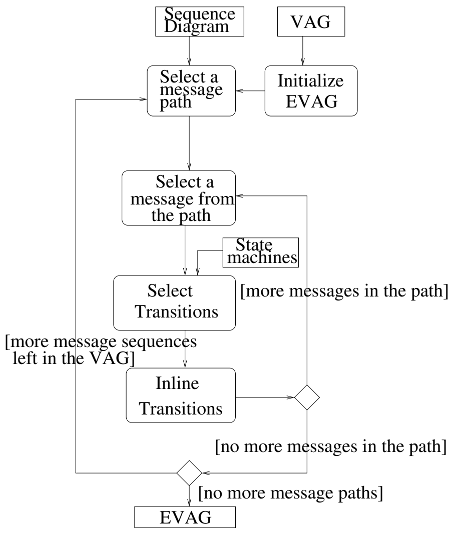


Figure 5. EVAG Generation from VAG

The activity diagram in Figure 5 describes the EVAG generation approach. First, we initialize the EVAG to be a clone of the VAG. Then we consider all the message paths in the corresponding sequence diagram, one at a time. Currently, if there are loops, we consider each loop twice, once with zero and then with one traversal through the loop. We examine each message one by one in each path. From the state machines, we select a set of transitions that can result from a message and are relevant to the path. This process is called *Transition Selection*. We keep modifying the EVAG based on the selected transitions. This process is called *Transition Inlining*. Modifications resulting from all the messages in all the message paths eventually result in the generation of the complete EVAG.

We use state machines from the video store example to illustrate the steps in our approach. The state machines describing the behavior of *Copy* and *MembershipAccount* are shown in Figures 6 and 7 respectively. Figure 8 shows the EVAG that is obtained from the VAG in Figure 3.

3.1. Transition Selection

Transition selection for a message refers to the process of identifying transitions that the message can fire in the state machine of its recipient object. We select the transitions that are valid for the message in the context of the path under consideration. In the following steps, we describe how we identify such valid transitions.

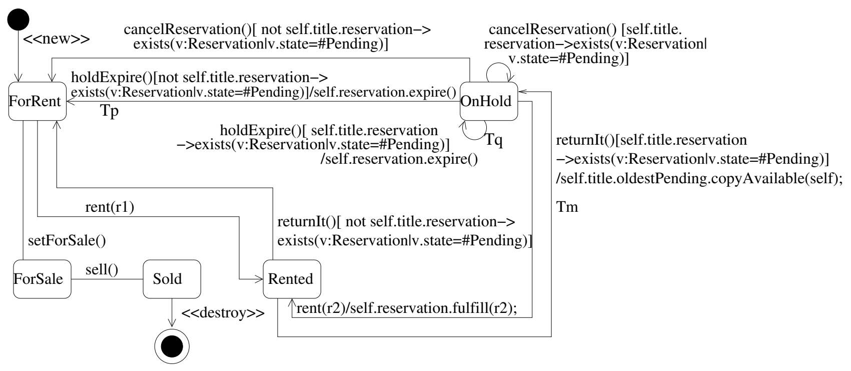
Step 1: Consider a path $< m_1, m_2, \ldots, m_n >$. Suppose that we need to select the transitions for some m_i in the path. For the first message, m_1, in the path, the target object can be at any state before m_1 was sent. Therefore, we select any transition fired by m_1 from any state in the state machine.

For all other m_i, where $(i > 1)$, we identify all the states that can result from m_{i-1} in the state machine of the target object of m_i. We select all the transitions fired by m_i from those states.

Consider, for example, message 2 (`updateDues()`) during the execution of the path 1-2-3-4-5-6-7-8-9, in the sequence diagram in Figure 2. The target of message 2 is `account`, an instance of *MembershipAccount*. Just before message 2 is sent, *account* can only be in the *Active* state. This is because *Active* is the only state where the event `updateDues()` is legal and the preceding message, `returnRental()` (message 1), does not affect the state of the *account* object. In the state machine of *MembershipAccount* in Figure 2, there are three transitions fired as a result of the message `updateDues()` from the *Active* state. They are Ta and Tb. We select both the transitions.

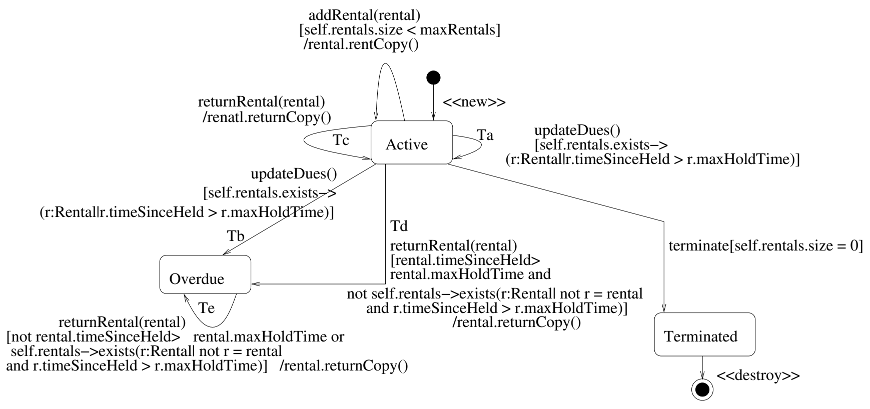
If message m_{i+1} exists, it can impose additional constraints for selecting transitions. We use the constraints to further filter the set of transitions selected in step 1. We consider two types of relationships that message m_i may have with m_{i+1}. In both cases, the target object of m_{i+1} must be state-dependent and m_{i+1} must fire some transition in the target objects' state machine. We now describe the steps for selecting transitions in the two cases.

Step 2: The target object of m_i is the source of m_{i+1}.



<<new>>
cancelReservation()[not self.title.reservation–> exists(v:Reservation|v.state=#Pending)]
cancelReservation() [self.title. reservation–>exists(v:Reservation| v.state=#Pending)]
holdExpire()[not self.title.reservation–> exists(v:Reservation|v.state=#Pending)]/self.reservation.expire()
ForRent
OnHold
Tp
holdExpire()[self.title.reservation –>exists(v:Reservation|v.state=#Pending)] /self.reservation.expire()
Tq
returnIt()[self.title.reservation –>exists(v:Reservation|v.state=#Pending)] /self.title.oldestPending.copyAvailable(self);
Tm
rent(r1)
returnIt()[not self.title.reservation–> exists(v:Reservation|v.state=#Pending)]
setForSale()
ForSale
sell()
Sold
Rented
rent(r2)/self.reservation.fulfill(r2);
<<destroy>>

Figure 6. State Machine for the Copy Class from Briand et al. [3]



addRental(rental) [self.rentals.size < maxRentals] /rental.rentCopy()
returnRental(rental) /renatl.returnCopy()
<<new>>
Tc
Active
Ta
updateDues() [self.rentals.exists–> (r:Rental|r.timeSinceHeld > r.maxHoldTime)]
updateDues() [self.rentals.exists–> (r:Rental|r.timeSinceHeld > r.maxHoldTime)]
Tb
Td
returnRental(rental) [rental.timeSinceHeld> rental.maxHoldTime and not self.rentals–>exists(r:Rental| not r = rental and r.timeSinceHeld > r.maxHoldTime)] /rental.returnCopy()
terminate[self.rentals.size = 0]
Overdue
Te
returnRental(rental) [not rental.timeSinceHeld> rental.maxHoldTime or self.rentals–>exists(r:Rental| not r = rental and r.timeSinceHeld > r.maxHoldTime)] /rental.returnCopy()
Terminated
<<destroy>>

Figure 7. State Machine for the MembershipAccount Class

Thus, the operations resulting from m_i and m_{i+1} have a caller-callee relationship. The operation invoked as a result of m_i results in the sending of m_{i+1}.

We examine the state machine of the target object of m_i. We select the transitions occurring as a result of m_i according to step 1. We filter the set of transitions by selecting only those that generate m_{i+1} as an action.

For example, consider the pair of consecutive messages, `returnRental()` and `returnCopy()`, in the path, 1-2-3-4-5-6-7-8-9, in the sequence diagram in Figure 2. The operation, `returnRental()`, calls `returnCopy()`, and thus the messages have a caller-callee relationship. The target object of `returnRental()` is *account*. From the state machine in Figure 7, we find that during the execution of the message path and just before the message `returnRental()` is sent, *account* can be in two states: *Active* and *Overdue*. There are three transitions, Tc, Td, and Te, from the states, `Active` and `Overdue`, that are fired by the call `returnRental()`. We select these two transitions. Following step 2, we look for a call, `returnCopy()`, as an action of some transition. We find that `returnCopy()` is called as an action on all the transitions that are fired by `returnRental()`. Therefore, all the three transitions Tc, Td, and Te are selected.

Step 3: The target objects of m_i and m_{i+1} are the same irrespective of their source objects. The messages can be the call events of two consecutive transitions in the state machine of their target object. We select the transitions occurring as a result of m_i according to step 1. We filter the transitions by selecting only those that are immediately followed by a transition fired as a result of receiving m_{i+1}.

The pair of consecutive messages, `updateDues()` and `returnRental()` (messages 2 and 4), in the sequence diagram in Figure 2 bears such a relationship because they have the same target object, `account`. As illustrated in Step 1, we select the transitions Ta and Tb for the message `updateDues()`. Now we verify if the selected transitions are followed by any transition fired by `returnRental()` in the state machine of *MembershipAccount* class. We find that Ta is followed by Tc and Tb is followed by Te in the in Figure 7. Both Te and Tc are fired by `returnRental()`. Therefore, both Ta and Tb satisfy the condition described in Step 3 and both of them are selected for the message `updateDues()`.

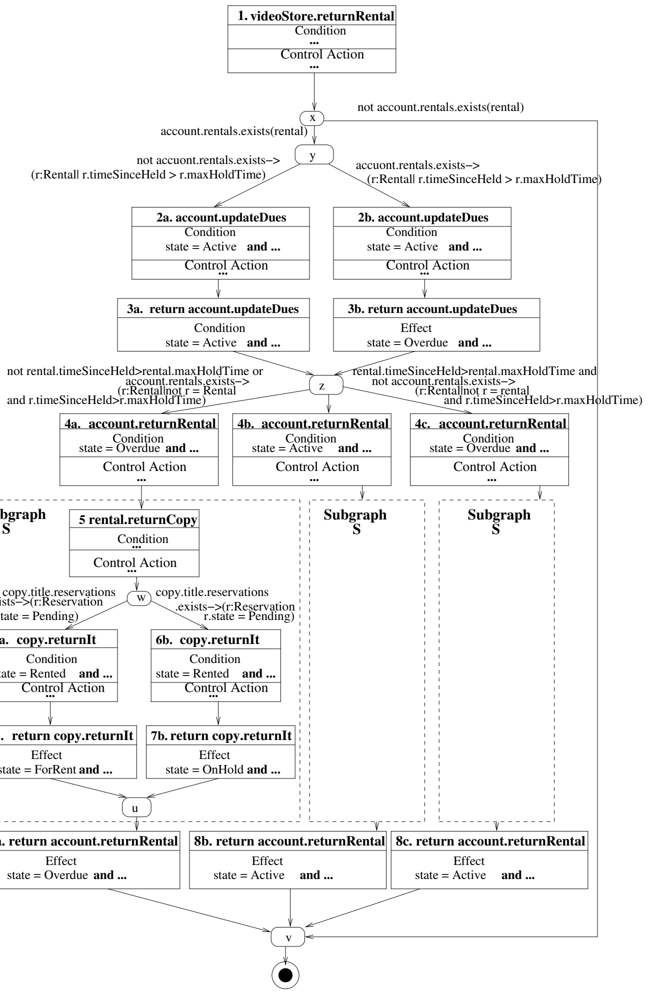

1. videoStore.returnRental
Condition
Control Action
x
not account.rentals.exists(rental)
account.rentals.exists(rental)
y
not accuont.rentals.exists–> (r:Rental| r.timeSinceHeld > r.maxHoldTime)
accuont.rentals.exists–> (r:Rental| r.timeSinceHeld > r.maxHoldTime)
2a. account.updateDues
2b. account.updateDues
state = Active and ...
3a. return account.updateDues
3b. return account.updateDues
Effect
state = Overdue and ...
z
not rental.timeSinceHeld>rental.maxHoldTime or account.rentals.exists–> (r:Rental|not r = Rental and r.timeSinceHeld>r.maxHoldTime)
rental.timeSinceHeld>rental.maxHoldTime and not account.rentals.exists–> (r:Rental|not r = rental and r.timeSinceHeld>r.maxHoldTime)
4a. account.returnRental
4b. account.returnRental
4c. account.returnRental
Subgraph S
5 rental.returnCopy
w
not copy.title.reservations .exists–>(r:Reservation r.state = Pending)
copy.title.reservations .exists–>(r:Reservation r.state = Pending)
6a. copy.returnIt
6b. copy.returnIt
state = Rented and ...
7a. return copy.returnIt
7b. return copy.returnIt
state = ForRent and ...
state = OnHold and ...
u
8a. return account.returnRental
8b. return account.returnRental
8c. return account.returnRental
v

Figure 8. Video System EVAG

3.2. Transition Inlining

Transition *inlining* refers to the process of expanding a message node in a VAG to expose the transitions the message can possibly fire. During this process, we inline the transitions selected earlier into the corresponding VAG node to construct the EVAG.

We illustrate transition inlining by taking an example of a simple sequence diagram shown in Figure 9(a). Figure 10(a) shows a part of the VAG obtained from the sequence diagram and it contains the nodes for the message m. It shows the node representing the message m along with a `predecessor` node in the VAG. The call node is followed by a subgraph containing all the VAG nodes representing messages that are directly or indirectly called by m.

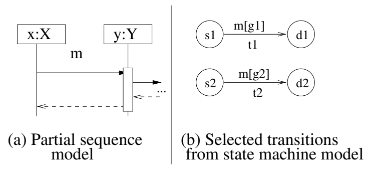

x:X
y:Y
m
(a) Partial sequence model
s1
m[g1]
t1
d1
s2
m[g2]
t2
d2
(b) Selected transitions from state machine model

Figure 9. Simple model

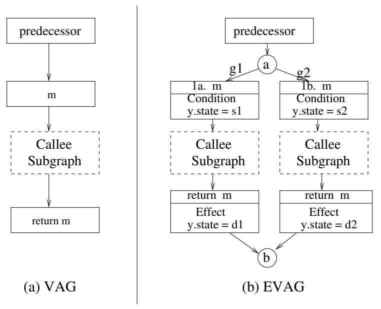


Figure 10. VAG and EVAG Obtained from Simple Model

The callee-subgraph is followed by the node representing the return message of m.

Figure 9(b) shows two transitions that are selected for m in the transition selection phase. The part of the EVAG obtained by inlining the two selected transitions into the VAG is shown in the Figure 10(b). The transformation consists of three different steps.

Split the Call Node: For every transition t selected for m, we make a copy of the call node of m. In the `Condition` section of every copy of the call node we add a constraint `(object.state = StateName)` using conjunction with the existing ones. The `StateName` refers to the source state of the transition t to which the copy of the call node corresponds. This condition specifies that the target object must be in the state, `StateName`, in order for the operation corresponding to m to be called on it and also result in the firing of t. In this example, there are two transitions and we generate two copies of the call node.

The generated copies of the call node are connected to the predecessor node by introducing a control node a, connecting the predecessor directly to a and then creating edges from a to each of the call node copies. The edge from a to a call node copy is associated with the guard of the transition the call node copy corresponds to. In Figure 10(b), call nodes `1a` and `1b` are created for transitions $t1$ and $t2$ in Figure 9(b). The edges `a---1a` and `a---1b` are annotated with guards $g1$ and $g2$ of transitions $t1$ and $t2$ respectively.

Split the Return Node: Similarly, we also split the VAG node that represents the return message of the operation call. For every transition t selected for m, we make a copy of the return node of m. In the `Effect` section of every copy of the return node, we add a constraint `(object.state = StateName)` as a conjunction of the existing constraints. `StateName` refers to the destination state of the transition t. The constraint specifies that the target object must be in the state `StateName` after the operation call fires the transition t and returns. Figure 10(b) shows two copies of the return node for the two selected transitions.

Duplicate the Callee Subgraph: To connect every pair of corresponding copies of call and return nodes, we duplicate the callee subgraph and insert it between the nodes. In this example, we generated two copies of the callee subgraph to connect the two pairs of call and return nodes.

We apply these rules to generate the EVAG in Figure 8 from the VAG in Figure 3. The ellipses in the nodes refer to the constraints in the original VAG nodes, which were copied to the corresponding EVAG nodes. The complete constraint is not shown due to limited space. As described earlier, we select the transitions Tc, Td and Te for the message `returnRental()`. To inline the transitions, we make three copies of the original VAG call node of the message `account.returnRental()`. The copies are `4a`, `4b`, and `4c`, which represent calls to `returnRental()` that fires transitions Te, Tc, and Td respectively. The edge from the predecessor to the call node copy is annotated with the guard of the appropriate transition. For instance, the edge `z---4a` is annotated with the guard of transition Te.

Nodes `8a`, `8b`, and `8c` represent the corresponding copies of the return node for `returnRental()`. Each call node copy is connected to its corresponding return node copy by inserting a duplicated callee subgraph, S, of the message `returnRental()`. The conditions of the call node copies and the effects of the return node copies are augmented with the state constraints for the appropriate source and destination states.

3.3 Generating Test Inputs

Once we generate the EVAGs, we solve the path constraints to generate test inputs for each EVAG path. To our knowledge, there exists no constraint solver that can solve OCL constraints directly. We use Alloy [8] and its analyzer to specify and solve the path constraints. Alloy's declaration syntax makes it easier to specify the structural properties of object oriented systems. Corresponding to each path we generate an Alloy constraint program from the classes and their associated OCL invariants in the class diagram and the constraints associated with the path. We convert the classes to Alloy signatures, the associations to Alloy relations and the sequence diagram lifelines to singleton signatures. We convert all the OCL constraints to Alloy expressions according to the rules described in Anastasakis et al. [2]. The solution provided by the Alloy analyzer gives an object configuration, initial values of the attributes, and the arguments of the first message. We generate a JUnit test case from the solution. We build the test fixture by creating the object configuration and setting the attributes as obtained from the solution. The test case then calls the first operation of the sequence diagram under test using the solved values of the arguments. We transform the post-condition of each system operation under test into a JUnit assert statement and use this as our oracle.

Figure 11(a) illustrates a partial initial object configuration obtained by solving the constraints of the VAG path 1-a-2-3-4-5-6-b-end in Figure 3. The VAG path is split into 12 possible execution paths in the EVAG in Figure 8. Figure 11(b) shows the object configuration obtained by solving the constraint along the EVAG path 1-x-y-2b-3b-z-4a-5-w-6b-7b-u-8a-v-end. The object configuration contains an additional *Rental* instance, `rental1`, which is overdue (i.e., `timeSinceHold > maxHoldtime`) for the *account*. This satisfies the condition on the edge `y---2b`, thereby firing transition Tb in the state machine of *MembershipAccount*. It also contains a *Reservation* object `reservation`, in the `Pending` state. This enables the condition on the edge `w---6b`, thereby firing transition Tm, in the state machine of class *Copy* in Figure 6.

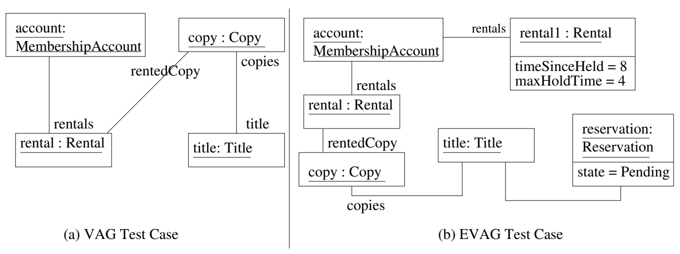


Figure 11. Partial Object Configurations

The all-EVAG-path coverage criterion subsumes the all-VAG-path coverage criterion. A VAG test input executing a message that fires multiple transitions will always fire some transition. For every such VAG test input, there is always an EVAG test input that is designed to fire the same transition.

3.4 Automation of the Approach

The approach is currently being implemented in a prototype tool. Generating a VAG from a sequence diagram involves converting control structures of sequence diagrams to a control flow graph. This is analogous to generating program control flow graphs from a high-level program, which has been already studied in the area of compilers and software engineering. VAG generation also involves deriving the constraints for each message node. Constraints are directly obtained from the models or derived in a straightforward fashion. This part has been automated.

Generation of an EVAG from a VAG is fully automatable as well and is currently work in progress. To extract the transitions resulting from a message, we need to determine the state of the target object immediately before the message is sent to it. We can obtain this information by keeping track of changes in the states of the target object while we traverse a message sequence. Under certain conditions (steps 2 and 3 of transition selection), we also resolve different types of relationships between two consecutive messages, such as caller-callee relationship. Relationship information can be automatically derived from the sequence diagrams. Transition inlining consists of well-defined rules of structural modifications of VAG with the selected transitions.

Algorithms for calculating a set of complete paths in a graph satisfying some coverage criterion already exist (see for example, [12]). We implemented one algorithm for the VAG and are now re-implementing it for the EVAG.

Generating JUnit test cases from an Alloy solution is possible as long a standard naming convention for the getter and setter methods of classes is followed. Creating the object structure in a JUnit test case requires us to invoke the getter and setter methods. Because the implementation may not be ready when the tests are generated, we have to assume that the implementation will follow some standard naming conventions for the methods.

4. Evaluation

We performed a pilot study to evaluate our test input generation approach on an implementation of the video store system. The system manages copies of videos for different titles. It also allows customers to create membership accounts. Customers having membership accounts can rent or return copies of videos and also make reservations for video titles that do not have an immediately available copy. Members can rent up to a certain maximum number of copies. Every rental has a due date, and if a member possessing a rental does not return it by the due date, the corresponding account becomes overdue. With an overdue account, one cannot rent any more copies.

We designed the system with nine classes: *Title*, *Copy*, *Reservation*, *MembershipAccount*, *Rental*, *Customer*, *Rental*, *VideoStore*, and *VideoStoreAdministration*. We created four sequence diagrams, each elaborating a system operation that represents a critical use case of the system. The system operations are `makeReservation()`, `cancelReservation()`, `rentCopy()` and `returnRental()`. We modeled the behavior of the classes `Copy`, `Reservation` and `MembershipAccount` as finite state machines.

Our implementation conforms to the design models we created. The implementations of the system operations follow the call sequences modeled in their respective sequence diagrams. We implemented the state-dependent behavior of *Copy*, *Reservation* and *MembershipAccount* as transition tables of their respective state machines, using conditional statements. The implementation of the domain layer of the system has a total of 863 lines of code.

We generated four VAGs from the sequence diagrams. We selected and inlined transitions from the three state machines into the VAGs to generate four EVAGs. For each of the EVAG and VAG paths we transform the path constraints into an Alloy constraint program. Some paths did not produce any test inputs because of inconsistent path constraints. This happened when path edges were associated with guard conditions that were inconsistent. Otherwise, there were no constraints that were not solvable by the Alloy analyzer. We obtained 31 test inputs, 12 from the VAG paths and 19 from the EVAG paths.

We used MuJava [10], an automated mutation system for Java programs, to generate mutants of our system and execute our tests on those mutants. We set up MuJava to apply all the mutation operators it supports. MuJava generated 179 mutants by applying the following class-level and traditional mutation operators. The description of the operators is taken from Ma et al. [10].

1. JDC: Java-supported default constructor creation.
2. PRV: Reference assignment with other comparable variable.
3. EAM: Accessor method change
4. JID: Member variable initialization deletion.
5. JTD: `this` keyword deletion.
6. JTI: `this` keyword insertion.
7. JSI: `static` modifier insertion.
8. ROR: Relational operator replacement.
9. COR: Conditional operator replacement.
10. COD: Conditional operator deletion.
11. COI: Conditional operator insertion.
12. LOI: Logical operator insertion.
13. AOIU: Unary arithmetic operator insertion.
14. AOIS: Short-cut arithmetic operator insertion.

Table 1 summarizes the results we obtained. For every mutation operator, all the mutants that are killed by VAG test inputs are also killed by EVAG test inputs. Table 1 shows that EVAG test inputs is considerably more effective than VAG test inputs in killing the mutants that are generated by mutating conditional, relational and logical operators (ROR, COR, COD, COI, and LOI). Many of these operators appear in the code that checks the transition guard conditions in the implementation of the different state machines. For a message that fires multiple transitions in a state machine, VAG test inputs executing that message

Table 1. Results of Mutation Analysis

Mutation Operator	Total Number of Mutants	Mutants Killed by VAG Test Inputs	Mutants Killed by EVAG Test Inputs
JDC	1	1	1
PRV	1	1	1
EAM	5	1	4
JID	1	0	0
JTD	3	2	2
JTI	3	2	2
JSI	18	7	9
ROR	28	16	24
COR	10	0	7
COD	6	2	5
COI	28	15	26
LOI	14	8	10
AOIU	9	6	7
AOIS	52	12	14
Total	179	73 (40.8%)	112 (62.6%)

cause only one of the many transitions to fire. EVAG test inputs result in firing all the transitions for such a message. The EVAG test inputs cover 26 out of the 33 transitions in the state machines while the VAG inputs cover 14 of them. This is one of the reasons why the EVAG test inputs are more effective in killing the mutants that target those transition guards.

For class-level mutation operators (JDC, PRV, EAM, JID, JTD, JTI, JSI), EVAG test inputs killed more mutants generated by some operators (EAM and JSI) than VAG test inputs. This happened because EVAG test inputs achieved higher coverage of class associations, attributes and methods. For the mutation operator JSI, although EVAG-based testing was more effective, both the test sets failed to kill many of the mutants. JSI mutates a class by declaring its attributes `static`. We can kill such mutants in a test case if we consider multiple instances of the class. However, the constraint solver created a single instance each for some of the classes. Therefore, JSI mutations applied to those classes remained alive after both test sets. EVAG paths sometimes resulted in more objects in the initial object configuration than VAG paths due to the additional constraints derived from the transition guards. This is the reason why some JSI mutations were killed by EVAG test inputs but not by VAG test inputs.

Both VAG and EVAG test inputs failed to kill most of the mutants that were generated by the arithmetic operator insertion (AOIS). We explain the reason with a concrete example. In our sequence diagrams, we have instances of conditional expressions that involve comparison of two integer variables, such as `timeSinceHeld > maxHoldTime`. While generating mutants from the implementation, AOIS arbitrarily applied arithmetic operators on the variables to generate mutants such as `(timeSinceHeld > --maxHoldTime)`. When we solved the path constraints that contain the condition `(timeSinceHeld > maxHoldTime)`, the constraint solver generated solutions by assigning values to the variables `timeSinceHeld` and `maxHoldTime`, so that the condition was satisfied. To kill such a mutant, we need close values for the variables `timeSinceHeld` and `maxHoldTime` (values differing by 1 in this case). However, the constraint solver generated widely separated values for the variables. Therefore, the insertion of arithmetic operators did not alter the value of the comparison and our tests failed to kill the mutants.

As explained in the previous example, given the same set of constraints, the effectiveness of our test inputs depends on the solution produced by the constraint solver. The solution that the Alloy solver produces is not always the same for the same constraint program. Also, to avoid large initial object configurations, we solved the constraints with the lowest bounds for the number of objects required to satisfy the constraints. We determine the lowest bounds by manual inspection of the constraints. The number used in the Alloy solver could affect the actual paths executed in the code because the number of objects can affect certain conditions and transition guards. Depending on which paths are executed in the code, certain faults may or may not be revealed.

5. Related Work

Abdurazik and Offutt [1] use collaboration diagrams for both static checking and test generation. They define a test criterion that requires all the messages in collaboration diagrams to be sent at least once.

Offutt and Abdurazik [13] describe a test generation approach based on UML state machines. They present algorithms to generate test inputs based on transition coverage, transition pair coverage and full-predicate coverage criteria.

Pilskalns et al. [14] propose a framework for generating test inputs by combining class diagrams and sequence diagrams. They convert each sequence diagram into a directed acyclic graph called OMDAG, which models the control flow of operation calls in the sequence diagram. They also construct an Object-Method Execution Table to capture method call execution sequences for each test case.

Lugato et al. [9] present the tool AGATHA, which validates specifications described by UML state machines and generates test cases from them as well. AGATHA validates specifications using exhaustive symbolic path coverage. Like our approach, it also generates test cases for symbolic paths by constraint solving. However, unlike AGATHA's exhaustive path coverage approach, we identify and test the state machines based on critical scenarios determined from the sequence diagrams.

Hartmann et al. [7] proposed an approach to generating test inputs from UML state machines that model the behavior of distributed component-based applications. State machines of individual components are combined and then used to build a test design, which identifies behavioral equivalence classes within the structure of the model. The test design is used to generate test cases.

Naslavsky et al. [11] use sequence diagrams to generate test inputs. A control-flow representation is used along with domain analysis of the parameters of the sequence diagram.

Briand et al. [5] proposed an approach to generate constraints on test data for testing a transition sequence selected from a state machine. From the transition sequence under test, they build a graph structure called Invocation Sequence Tree (IST). The authors use the IST structure to derive the test constraints that must by satisfied by any test input that executes the transition sequence under test.

All the above approaches use only one kind of behavioral UML model for test generation, either sequence diagrams or state machines. Our approach is novel in the sense that we combine the information from two types of behavioral models for the purposes of test input generation.

Briand et al. [4] present a system test methodology, TOTEM, to generate test requirements from UML use case diagrams, sequence diagrams and class diagrams. This is another example where models of multiple types are used just like ours, except that we are using more detailed state information when available.

6. Conclusions and Future Work

We proposed an approach for generating an EVAG structure that combines information from UML sequence and state machine models to generate test inputs. The pilot study showed that our approach improved test effectiveness by using more precise information about the effects of message sequences from state machine models.

The pilot study described in this paper is the beginning of a large scale validation activity to investigate the effectiveness of the approach. We will evaluate the cost of test input generation using our approach. We will extend our approach in order to support a wider range of UML features. In this work we assumed that all events in a state machine are call events. The presence of change events that fire when a boolean predicate becomes true poses a new challenge because a message can fire a transition and as a side effect can also enable the predicate of another change event. Moreover, we have only focused on synchronous operation calls in sequence diagrams. We will investigate other types of messages, especially asynchronous messages. We will also compare the cost and effectiveness of our approach with state machine based approaches.

References

[1] A. Abdurazik and A. J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the 3rd International Conference on the UML, York, UK, October 2-6, 2000*, volume 1939 of *LNCS*, pages 383–395. Springer, 2000.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. Uml2alloy: A challenging model transformation. In *Proceedings of the 10th International Model Driven Engineering Languages and Systems, Nashville, USA, 2007*, pages 436–450, 2007.

[3] A. Andrews, R. B. France, S. Ghosh, and G. Craig. Test Adequacy Criteria for UML Design Models. *Journal of Software Testing, Verification and Reliability*, 13(2):95–127, April-June 2003.

[4] L. Briand and Y. Labiche. A UML-based approach to system testing. In *Proceedings of the 4th International Conference on the UML*, pages 194–208, Toronto, Ontario, Canada, oct 2001.

[5] L. C. Briand, Y. Labiche, and J. Cui. Automated support for deriving test requirements from UML statecharts. *Software and System Modeling*, 4(4):399–423, 2005.

[6] T. Dinh-Trong, S. Ghosh, and R. France. A Systematic Approach to Generate Inputs to Test UML Design Models. In *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering*, pages 95–104, Raleigh, NC, November 2006.

[7] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-Based Integration Testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 60–70, New York, NY, USA, 2000. ACM.

[8] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[9] D. Lugato, C. Bigot, Y. Valot, J.-P. Gallois, S. Gérard, and F. Terrier. Validation and Automatic Test Generation on UML Models: the AGATHA Approach. *International Journal on Software Tools for Technology Transfer*, 5(2):124–139, 2004.

[10] Y. S. Ma, J. Offutt, and Y. R. Kwon. "MuJava: An Automated Class Mutation System". *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[11] L. Naslavsky, H. Ziv, and D. J. Richardson. Towards traceability of model-based testing artifacts. In *A-MOST '07: Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, pages 105–114, New York, NY, USA, 2007. ACM.

[12] S. C. Ntafos and S. L. Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, 5(5):520–529, 1979.

[13] A. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on the UML, Fort Collins, CO, USA, October 28-30, 1999*, volume 1723 of *LNCS*, pages 416–429, 1999.

[14] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. B. France. Testing UML Designs. *Information and Software Technology*, 49(8):892–912, 2007.