Generating test data to test UML Design Models *

Trung Dinh-Trong, Sudipto Ghosh, Robert France[1]
{*trungdt,ghosh,france*}*@cs.colostate.edu*
and Anneliese Andrews[2]
aandrews@eecs.wsu.edu

[1] Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
[2] Department of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164

Abstract. This paper presents an approach to generating inputs that can be used to test UML design models. A symbolic execution based approach is used to derive test input constraints from a Variable Assignment Graph (VAG), which presents an integrated view of UML class and sequence diagrams. The constraints are solved using Alloy, a configuration constraint solver, to obtain the test inputs.

Keywords: *design quality, test generation, testing models, UML, symbolic execution, test adequacy criteria*

1 Introduction

For model driven development approaches to succeed, there is a need to develop techniques for validating models. Studies show that many software faults occur in the design phase [1], hence, finding and removing faults in design models can lead to better quality software. Currently, UML design models are typically evaluated using walkthroughs, inspections, and other informal types of design review techniques that are largely manual and, consequently, tedious.

In our previous work, we proposed an approach to testing UML design models consisting of class diagrams, sequence diagrams, and activity diagrams. We gave a set of test adequacy criteria based on class and sequence diagram's elements [2], and described a technique to execute models under test with test inputs [3]. In this paper, we propose a technique to generate test cases that satisfy the sequence diagram criteria described in [2]. The technique uses a representation called a Variable Assignment Graph (VAG) that integrates relevant information from class and sequence diagrams. The VAG is used to derive test input constraints that are then solved by the constraint solver, Alloy [4].

* This research was supported in part by National Science Foundation Award #CCR-0203285 and an Eclipse Innovation Grant from IBM.

2 Model testing approach

A design model under test, DUT, consisting of class and activity diagrams, is transformed into an executable form, $EDUT$. The $EDUT$ is a program that simulates the behaviors modeled in the DUT. Test scaffolding is added to the $EDUT$ to automate test execution and failure detection. The result is referred to as the testable form of the design, $TDUT$. Test execution is performed by applying the test inputs to the $TDUT$. During test execution, the system behavior is recorded and observed in terms of changes in the system state, where a system state is represented as an object configuration. Failures are detected when the observed behavior differs from the expected behavior. For a more detailed description of the test execution approach, please see [3].

In the testing approach, testers select a set of test adequacy criteria [2]. A set of test inputs that satisfies the selected criteria is generated from the DUT. Each test input is generated based on one sequence diagam. In our previous work, a test input includes a sequence of operation calls that executes the test [3]. In the test input generation approach described in this paper, the sequence of operation calls contains only one element, which is the operation call that initiates the sequence diagram under test. Thus a test input in this paper is a tuple consisting of two components: the start configuration, S, and the set of parameter values, P. Before a test is performed, the $TDUT$ is brought to the start configuration, S. A system configuration is described as a set of objects, a set of links between the objects, and the set of object attribute values. Testing is performed by invoking the system operation that initiates the sequence diagram under test using the set of parameter values, P.

Our test case generation technique aims at deriving test inputs that satisfy sequence diagram based test adequacy criteria described in [2]. Three criteria were defined based on the coverage of conditions, predicates, and message paths in interaction diagrams:

1. *Condition Coverage* criterion: Testing must cause each condition in each decision to evaluate to both `TRUE` and `FALSE`.
2. *Full Predicate Coverage* criterion: Testing must cause each clause in every condition in the sequence diagram to take the values of `TRUE` or `FALSE` while all other clauses in the condition have values such that the value of the condition will always be the same as the clause being tested.
3. *All Message Path Coverage* criterion: Testing must cause each possible message path in the sequence diagram to be traversed at least once.

3 Test input generation issues

The design test adequacy criteria used in our approach are similar to control flow based programming test adequacy criteria [5]. In program testing, test inputs that satisfy control flow based criteria are usually generated using path based test case generation. In path based test generation approaches, execution paths that satisfy test adequacy criteria are first identified. Test cases are then derived

so that the paths are traversed. Generally, path based test generation approaches can be categorized into execution based and symbolic execution techniques [6].

Execution-based test generation [7] involves analyzing the execution of programs with actual inputs and iteratively refining the input values until a desired path is traversed. The program is first executed with an input, and the execution flow is monitored. When a branch that is not in the selected path is executed, function minimization search algorithms are used to find an input value so that the desired branch is traversed. Execution based test generation requires excessive execution, especially when the chosen path is not traversable.

In symbolic execution techniques [8], programs are executed using symbolic values of variables instead of actual values (such as numbers). As a result, every branch predicate along the path is expressed in terms of the input symbols. Symbolic evaluation is used to generate a set of equalities and inequalities involving the program input values, which must be satisfied for the path to be traversed. Constraint solvers (such as e-box consistency based solver [6]) can be used to solve the inequalities and find a solution that serves as a test input.

The existing path based test generation approaches assume that programs are described declaratively. In UML models, behavior can be described both declaratively and imperatively. While sequence diagrams are specified imperatively, they only describe the interaction between objects (e.g., the method calls and the creation of objects). What happens inside each object (e.g., the modification of attribute values) can be specified declaratively in operation post-conditions.

Existing path based approaches have been developed for procedural programs. In existing symbolic execution techniques, the test inputs, which are either program parameters or variables that are defined using input statements, always have primitive types (such as *boolean*, *integer*, *real*, and sometimes, *array*). Thus, these techniques lack a mechanism to model system configurations, which are part of the test inputs in our testing approach. In our approach, inputs may be (1) the parameters of the operation that initiates the sequence diagram, and (2) the variables that are used to define the start configuration.

Another challenge in generating test inputs from UML models is that models are described using various diagram types, where each type only captures a partial view of the system. Class diagrams capture the structural view of the system, sequence diagrams capture the interactions between objects, and the pre- and post- conditions in class diagrams capture the effect of each operation. Generating test inputs that satisfy the criteria described at the end of Section 2 requires analyzing the sequences of messages exchanged between objects and the effect of executing the action associated with each object. The sequence of messages, as well as the effect of executing actions associated with *create* and *destroy* messages are specified only in sequence diagrams. The effect of executing the action associated with the *call operation* messages, however, is captured only in operation pre- and post-condition in class diagrams. Hence generating test inputs from UML models requires a mechanism that combines relevant information in class and sequence diagrams.

4 Generating test inputs using constraint solvers

To generate test inputs from UML class and sequence diagrams, we first integrate the UML class and sequence diagrams into a directed graph called a Variable Assignment Graph (VAG). A VAG combines the relevant information, which is needed for test input generation, from UML class and sequence diagrams. A VAG records how and when variables are defined and used in a sequence diagram. It also records the conditions that enable the sending of messages in the sequence diagram. VAGs have the form that is similar to the control flow graph in code level. Hence, existing control flow graph based path generation technique, such as [9], can be applied to VAGs.

4.1 Variable Assignment Graph (VAG)

A VAG is composed of nodes and directed edges. The nodes record the changes in values of variables during the execution of a sequence diagram and the conditions necessary for the changes to occur. The edges represent control flow.

There are two types of nodes: message nodes and control nodes. Each message node, denoted by a rectangle, is derived from a message in the sequence diagram. A message node is composed of three parts: *Condition*, *Control action*, and *Effect*. The *Condition* part records the configuration constraints that enable the sending of messages. Such constraints include the existence of the recipient of the message and the link between the sender and the receiver objects. The *Control action* part records the assignment of the actual parameter values to the formal parameters. The *Effect* part records the changes in variables after the execution of an operation call. These changes include object creation and destruction, as well as state variable updates as specified in operation post-conditions. Any part of a node can be empty. The *Condition* part is empty if the corresponding message is a return message, since the existence of the receiving object and the link is implied by the call message, which is already sent before the return message. The *Control action* part is empty when a message does not have any parameter. The *Effect* part is empty when there is no change in the configuration when the message is received.

VAG control nodes are denoted by rounded boxes and are used to represent(1) merging and branching of path, (2) loops control, and (3) the termination of execution.

VAG edges are denoted by directed lines. An edge can be associated with a branching predicate in the sequence diagrams. Sequence diagram branching predicates include message conditions, conditions associated with alternative fragments, and conditions associated with loop fragments. The branch predicates in a VAG are shown by text displayed next to the corresponding edges.

Figure 1 shows an example of a UML design model (Figure 1(a)) and its corresponding VAG graph (Figure 1(b)). In this example, the *ATM* system handles a "withdraw" request by first checking the *balance* in the correspond *account*. If the *balance* is greater than the requested withdrawal *amount*, the transaction will be executed and the *balance* will be deducted. Messages 3 and 5 are return

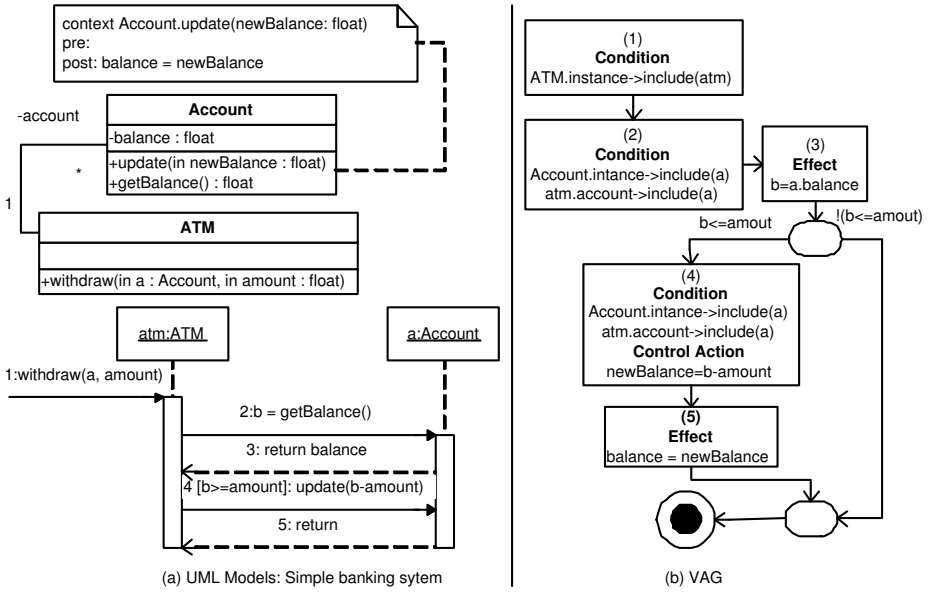

Fig. 1. A UML model and its corresponding VAG

messages, denoting that the operation calls associated with messages 2 and 4, respectively, are returned.

The k^{th} VAG node in Figure 1(b) is derived from the k^{th} message in the sequence diagram. For example, node 4 is generated from message number 4, a call message. The *Condition* part inside node 4 states that message 4 can be sent if `a` exists, and there is a link between instances, `atm` and `a`. The *Control Action* part inside node 4 records that the value of the actual parameter, `b-amount`, is assigned to the formal parameter, `newBalance`. Node 5 is generated from the return message 5. The *Effect* part inside node 5 records that the state variable, `a.balance`, is updated with the new value, `newBalance`. This information is derived from the post-condition of the operation `Account::update(float)`.

Because of the space limitation, we do not describe how UML models are transformed into VAGs.

4.2 Generating test inputs from VAG

A sequence diagram based test adequacy criterion defines a set of sequence diagram structures that need to be covered during testing. A set of paths in the VAG needs to be selected such that the execution of these paths guarantees that all the desired structures in the corresponding sequence diagram are traversed at least once. Since each message in a sequence diagram is transformed into a VAG node, it is easy to see that a set of paths that traverses all nodes in the VAG

satisfies the "*Each Message on a Link*" criterion. A set of paths that traverses all VAG edges satisfies the "*Condition coverage*" criterion. The set of all VAG paths satisfies the "*All Message Paths*" criterion. Finding the set of paths that covers all VAG nodes and edges is similar to finding paths that cover all statements or branches in a program. Existing path generation techniques, such as those using dominance and implication graphs [9], can be applied.

With a few modifications, symbolic execution can be applied to a VAG to generate path constraints. Traditional symbolic execution techniques use branching conditions and the assignment statements to form the path constraints. In UML models, only assignment statements that assign return values of method calls to variables are explicitly specified in sequence diagrams, e.g., messeage number 2 in Figure 1(a). The other assignments, which happen inside the operations, are implicitly specified in operation post-conditions. Hence, in our approach, a path constraint must include changes that are stated in the post-conditions. Moreover, path constraints in our approach must also include the configuration constraints that enable the sending of messages. The path constraints can be constructed by forming the conjunction of the following conditions:

1. All the conditions in the *Condition* part.
2. All the branch predicates.
3. All variable definitions in the *Control Action* and *Effect* parts.

For example, the constraint for the path *1-2-3-END* is as shown in Figure 2(a).

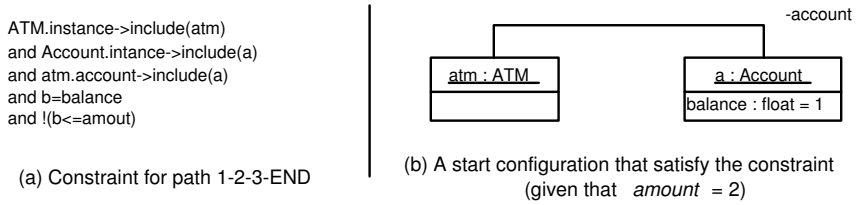

Fig. 2. The constraint and a solution for the path 1-2-3-END

The path constraints produced in our test generation approach contain numeric symbols (e.g., `i_0`) as well as symbols that represent system configurations (e.g., `pc` and `pc.categories`). Hence our constraint satisfaction problem cannot be solved by pure numerical constraint solvers (e.g., the e-box consistency based constraint solver [6]).

Alloy [4] allows the specification of constraints that contain integer and configuration symbols. We specify the path constraints, which are originally specified using the OCL, using the Alloy constraint language. Since the input configurations need to satisfy the class diagram constraints, we also transform the class diagram using the Alloy constraint language.

We solve our constraints using Alloy, a configuration constraint solver. Generally, the domain of configurations that a solver needs to search is infinite. However, Alloy can solve the constraint problems if we restrict the search range by setting a maximum total number of objects in the solution. In such cases, Alloy will either give us a solution, or report that there is no valid solution within the range. When we use Alloy to solve the constraint in Figure 2(a), with the maximum number of objects set to 2, we get a solution with `amount` = 2, and a start configuration as shown in Figure 2(b).

5 Conclusions

We outlined a systematic approach to generating inputs to test UML design models. We are currently conducting empirical studies to evaluate the effectiveness of the approach. The performance of a symbolic execution approach depends primarily on the size of the path constraints. We plan to study the growth of the constraints in our approach when the size of the model grows. Since design models are specified in less detail than the programs, we expect that in general the design level path constraints will be simpler than program level path constraints. We are also developing a prototype tool that automates the test input generation described in this paper.

References

1. Pressman, R.: Software Engineering - A Practitioner's Approach. 7th edn. McGraw-Hill, New York, NY (2001)
2. Andrews, A., France, R., Ghosh, S., Craig, G.: Test Adequacy Criteria for UML Design Models. Journal of Software Testing, Verification and Reliability **13** (2003) 95–127
3. Dinh-Trong, T., Kawane, N., Ghosh, S., France, R., Andrews, A.: A Tool-Supported Approach to Testing UML Design Models. In: Proccedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS). (2005)
4. MIT Laboratory for Computer Science: Alloy. `http://alloy.mit.edu/` (2005)
5. Myers, G.J.: The art of software testing. John Wiley & Sons (1979)
6. Tran Sy, N., Deville, Y.: Consistency techniques for interprocedural test data generation. In: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, Helsinki, Finland (2003) 108–117
7. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering **16** (1990) 870–879
8. Boyer, R.S., Elspas, B., Levitt, K.N.: Select–a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software, Los Angeles, CA (1975) 234–245
9. Bertolino, A., Marre, M.: Automatic generation of path covers based on the control flow analysis of computer programs. IEEE Transactions on Software Engineering **20** (1994) 885–899