JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2009

Vol. 09, No. 2, March–April 2010

Automatic Test Data Synthesis using UML Sequence Diagrams

Ashalatha Nayak and **Debasis Samanta**
School of Information Technology
Indian Institute of Technology, Kharagpur
Kharagpur, West Bengal, India
{asha_nayak1@yahoo.com,debasis.samanta.iitkgp@ gmail.com}

Model based testing techniques are used to generate test scenarios from a behavioral description of system under tests. For a large and complex system, there are usually a large number of scenarios and hence a large number of test paths also called test specifications. To automate test execution, each test specification should be augmented with appropriate test input data. In this paper, we propose an approach of synthesizing test data from the information embedded in model elements such as class diagrams, sequence diagrams and OCL constraints. In our approach, we enrich a sequence diagram with attribute and constraint information derived from class diagram and OCL constraints and map it onto a *structured composite graph* called SCG. The test specifications are then generated from SCG. For each test specification, we follow a constraint solving system to generate test data.

1 INTRODUCTION

Software testing is an expensive process in a software development life cycle and remains the primary activity to achieve confidence and quality in the developed software. The testing involves running an implementation against input selected by the test design and evaluating the response [1]. The goal is to reveal faults by exercising the software on a set of test cases. The test design phase mainly focuses on test case generation to derive test paths and test data generation to derive test inputs. Testing techniques can be classified into functional techniques and structural techniques. The functional technique uses the functional specification of a program rather than the implementation of the program to derive test cases. On the other hand, structural techniques derives test cases by exercising different paths in an implementation.

On the basis of the scope and extent of the implementation being considered, the above testing techniques can be focused to unit tests, integration tests and system tests. Testing at the system level is concerned with finding discrepancies between the actual behavior of the system and the desired behavior described on the specifications [2]. System tests differ from unit and integration tests in that system tests focus on the functionalities of applications as black boxes rather than methods or objects as white boxes. Unit testing considers an object as a unit to be tested and its behavior is tested. There exist automated frameworks such as JUnit [3] for this purpose. To support automation of system tests, several frameworks have been developed such as JSystem, JWebUnit etc. ([4], [5]) which cater to application specific testing such as testing of web applications, testing of GUI etc. Analogous to JUnit, however, there exists no framework for system testing of general purpose applications.

Typically, the sources of system testing information can be derived from different system representations those are available at the system level such as features used in product documentation, help screens etc. In object-oriented systems development, use cases which represent system scope capabilities [1] are widely considered as representations of system requirements. Of late, UML 2.0 has introduced a number of diagrams to represent systems at different level. Out of which use cases related diagrams are most suitable diagrams to specify system requirements. Here, each use case is expressed as a set of scenarios denoting main and alternative scenarios. Sequence diagrams also called interaction diagrams represent these scenarios as possible sequences of message exchanges among the objects to specify tasks. UML 2.0 sequence diagrams combines multiple scenarios by means of combined (interaction) fragments. A combined fragment may contain another combined fragment. This mechanism enables complex scenarios to be specified in a single sequence diagram. There are different types of combined fragments in UML 2.x such as repetition (loop), selection (alt/opt/break) and concurrencies (par) [6]. Execution of a combined fragment is controlled by means of an operator called interaction operator. A number of operands are there in a fragment which is groups of message sequences. To facilitate scenario representations and their flow analysis, we transform a sequence diagram into an intermediate form which unambiguously and in a structured way shows all the interaction operands and flow of control of these operands. In other words, this intermediate form resembles with that of conventional control flow graph [7]. The testable model allows us to trace a scenario corresponding to each message sequence path. Typically, a scenario in an interaction diagram is considered as a test path to test a system. Each scenario can thus be considered as an abstract representation of a test case. The actual test cases must contain specific values for the parameters to be able to find errors in a scenario execution. The test data generation phase addresses this translation in which test data is synthesized for each scenario (an abstract test case).

The approach proposed in this paper addresses mainly two issues: scenario generation and test data generation. The scenario generation concerns the generation of scenarios from the testable model of the sequence diagram. The problem of test scenario generation is to cover all paths with a constraint to limit explosion of paths, which arise due to loops and concurrencies. On the other hand, the test data generation identifies input data to execute the scenarios. The problem of test data synthesis is to identify input data satisfying a given test coverage criterion [8]. The input data for a scenario has to be chosen in such a way that the set of method invocations and constraints within the scenarios must take appropriate values to execute. Solving these constraints, it is possible to extract the test data. Hence, the test data generation method proposed in this paper includes three steps: (1) deriving constraints for the specified scenario, (2) solving the constraints along the scenario and (3) generating test data for finding test input to the variables involved in the scenario.

The contributions of this paper are as follows: First, a precise model known as *Structured Composite Graph* is proposed that builds information from sequence diagram, class diagram and OCL constraints. Second, a scenario oriented test data generation scheme is developed for generating test specifications and test data. This paper is structured as follows. An overview of related work is described in Section 2. A brief discussion on basic definitions and concepts used in our methodology is given in Section 3. Section 4 presents our proposed approach to test data generation. Section 5 presents an illustration of the approach. Finally Section 6 concludes the paper.

2 RELATED WORK

Test data synthesis is an essential task while generating test cases from model based specifications particularly in the context of automatic software testing. However, works concerned with test data generation from UML models are just beginning to emerge. This section presents the state of the arts on test data generation from programs followed by test data generation from model artifacts.

Test Data Synthesis from Programs

Test data generation methods ([9], [10], [11], [12], [13], [8], [14]) have been widely applied to unit testing and module testing in function oriented programming. The methods employ structural testing criteria to extract required information from programs. Depending on whether the control or data flow aspects of an implementation is chosen, the structural testing criteria can be classified as control flow and data flow based criteria. The control flow based criteria analyze the control flow such as statement, branch, loop and concurrent constructs of an implementation ([15], [16], [17], [18]). For this purpose, the control flow of a program is usually represented by a control flow graph, where the nodes are either a decision node or a node representing single entry and single exit sequence of statements known as block node. The edges represent possible control flows between nodes. On the other hand, the data flow based testing criteria require the data flow associations between definitions and uses of variables to be exercised [19]. For a given test criterion, the test data generation methods find a test input to exercise various test requirements. For example, statement coverage finds a test input for each program statement, branch coverage finds an input such that the execution traverses a specified edge of the control flow graph associated to the program and path coverage finds the input causing the execution of a specified path. Three different approaches have been proposed for automatic test data generation in this context.

Random test data generation

Random test data synthesis methods generate test input values using a random number generator. Although it is simple to generate random values, the generated values may not satisfy specific test requirements. It is therefore difficult to produce adequate test suites using this method. Hence, the usage is limited to evaluation of test requirements. For example, the approaches proposed by Korel [9] and Michael et al. [10] have used random test input generation as a baseline for evaluation of their test data generation methods.

Path oriented test data generation

The path oriented test data generation methods identify a set of paths and then generate input to execute the selected paths. For these methods, test generation is based on *symbolic execution* ([11], [12], [13]) or *dynamic execution* [8]. The symbolic execution methods ([11], [12], [13]) assign symbolic values to variables. For example, an input variable x is assigned a symbolic value such as $x = x_0$. After a few executions down the path, the value of the variable becomes a complex expression. On each branch predicate along the path, the predicate is expressed in terms of input variables of the program. The result is a constraint on the input variables to be satisfied by the test input. This kind of symbolic execution is repeated for all branch predicates along the selected path and a system of constraints is derived. Finally, the set of constraints are solved using constraint solving techniques which finds values for each input variable such that all the constraints hold. If values cannot be found satisfying the constraints then the path is declared infeasible and the next path is selected. Although many programming constructs such as loops, arrays, pointers etc. have been found very difficult to execute symbolically, the symbolic execution has been found to detect path infeasibility while solving the constraints.

The dynamic execution method, on the other hand, is based on actual execution of a program under test and test data is developed using actual values of input variables. In the beginning, all input variables are set to take some initial values. When the program is executed with this data, the flow is monitored which helps in determining whether the test requirements are satisfied or not. For example, for the branch predicate ($x > 10$), the goal of achieving a TRUE branch is to observe the value of variable when the execution reaches this condition. If the intended path is not taken, then the flow is altered by changing the values of input variables until the path is taken into account.

In the approach proposed by Korel [8], dynamic test generation is realized as a function minimization problem and the solution is obtained by minimization of values for input variables. In their approach, for a given program path, if the initial input is the solution then the main goal is achieved. If the initial input is not the solution, then the problem is expressed in terms of solving subgoals repeatedly until the main goal is reached or one of the subgoal cannot be solved. Two types of search strategies are employed for minimization - exploratory search to achieve the smaller moves and a pattern search to take the larger moves. The basic search procedure is to select one variable at a time and alter its value until the solution is found. The selected variable is altered by exploratory search and if the direction to proceed is correct, then the larger move on the desired direction is taken. In addition, dynamic data flow analysis is used to speed up the search process by determining the input variables that have a direct influence over the evaluation of the branch function. If the selected path is infeasible, then the efforts may go vain in performing the minimization.

Goal oriented test data generation

The goal oriented test data generation procedure is employed to overcome the difficulty of path execution methods in deciding path feasibility. For the selected node in a program known as goal node g, the procedure finds program input so that the node g will be executed irrespective of the path taken. Thus the path selection procedure is eliminated, instead a goal node is selected. To guide the search process in reaching the goal node, a heuristic approach is employed in chaining approach [14]. The chaining approach makes use of data flow analysis to identify a chain of nodes which are required to be executed before reaching goal node. If an undesirable execution flow is observed at a certain branch in the program, then the function minimization procedure is used to alter the execution flow at this branch. Although the approach succeeds in handling all types of programming constructs such as arrays and pointers, it may require a large number of executions of the program.

Test Data Synthesis from UML Models

Pilskalns et al. [20] present an approach for testing UML design models merging information from sequence diagrams and UML structural views. The test generation, execution and validation is addressed in their approach. They develop an aggregate model deriving information from the sequence diagram and combining with class information to build an instance of the class. The test adequacy criteria for both class diagram and sequence diagram is applied to their model. The test data generation uses Binder's domain analysis approach [1] for identifying the set of variables occurring in conditional expressions. Based on these variables, they create partitions for variables and select test values. The test cases are separately not formed and arise as a result of domain testing. The effectiveness of domain testing, however, is shown to be limited since it lacks a path selection criterion [21]. As a result, paths with faults may be skipped from domain testing. Another approach for testing UML design models is proposed by Dinh-Trong et al. [22]. The technique uses a *Variable Assignment graph* (VAG) by deriving information from a sequence diagram and its corresponding class diagram. The test input is generated based on the sequence diagram which is consisting of a start configuration and a set of parameter values. The design under test is brought to an initial state by the start configuration which is described as a set of objects and their attribute values. Two types of nodes are employed in VAG. A *message node* is for every message in the sequence diagram whereas *control node* denote branching, merging, looping and termination of execution. The post condition of the operation from the class diagram is used for specifying the changes in variables after the execution of an operation call. This information is maintained in the message node of a VAG. The symbolic execution technique is used for solving path constraint to generate test inputs. The major problem with their approach is the limited set of fragments. More specifically, multiple fragments and their nested combinations results in complex path constraints. The development of symbolic values and solving of path constraints using these variables are complicated using symbolic execution method.

Samuel et al. [23] propose a technique for generating test cases from UML communication diagrams. From the tree representation of the communication diagram, test cases (message paths) are selected as a post-order traversal of the tree in their approach. The test data is then solved by employing Korel's function minimization technique [8] on predicates selected from the messsage paths. Only behavioral information can be extracted from the communication diagram and therefore information concerning data type, attribute constraints, domain of variables etc. is not available for test data generation in their approach. Consequently, test data for different data types is not addressed and integer data type is considered as a default data type. Due to this limitation, the generated test cases may not be precise and need to be augmented with additional information before applying to test a target system.

Weileder et al. [24] develop an approach to deal with model based test generation using state machines, class diagrams and OCL expressions. A test tree is developed by connecting state machines and class diagrams from referencing transitions to operation calls. For this, each event in the state machine is interpreted as call event. In their approach, classification of the variables in OCL expressions is provided by recognizing the variables that can alter the value of attributes from those variables that cannot alter the value of attributes. The test generation process identifies variables and then creates partitions of the value ranges for all input variables. A test path is created as a test input sequence from the root to one of the leaves of the tree. A test case is then created by selecting representative value for each parameterized event of the path. In contrast to state machines, our approach uses sequence diagram to model the message exchanges among the objects involved in the interaction. The state machines denote behavioral properties of a class and requires integration of state machines corresponding to several objects. Consequently, extending their approach to a large case study is a complex process.

Automated test data generation for the Test and Testing Control Notation version 3 (TTCN-3) is discussed in Dai et al. [25] approach using classification tree method. In their approach, input data for abstract test cases of TTCN-3 are generated by partitioning of the data domain. They discuss three different ways to create partition for equivalence classes for choosing one representative from each class. However, they rely on the user to provide classification and test data selection.

Hartmann et al. [26] address the problem of test data generation and execution of system tests from UML activity diagrams. The category partition method is considered as the underlying test data generation technique for their approach. A category is defined for all the variables in the diagram. Accordingly, they discuss annotations to describe variables, partitions for variable ranges, coverage requirements etc. Based on these test requirements, equivalence classes in the system under test are identified. The test cases are created from the activity diagram by mapping its activities and transitions to partitions and choices such that all different paths of choices are covered. They provide evaluation of their experiments in comparison to the existing manual approach. However, the major concern with their approach is that the complex control structures such as concurrency, loops and their nested combinations are not addressed. In addition, simple predicates are evaluated as the branch predicates. In this respect, applying category partition to evaluate complex expressions is not explored.

Existing work consider test data generation for a given test path. However, none of these approaches define a precise model for the generation of test paths and then test data. Different types of faults may exist in different combinations of program statements. To reveal faults in all such combinations is a hard problem as number of statements and faults increase as the size of program grows. A testing strategy is needed to overcome this combinatorial problem. In this regard, we have defined an effective program testing strategy from the relevant model specifications. In addition, our approach can be used to expose all path oriented faults.

3 BACKGROUND

In this section, we begin with a discussion on UML 2.0 sequence diagrams, class diagrams and OCL constraints. Following this, we define *structured composite graph*, a directed representation of sequence diagram that combines information from the class diagram and OCL constraints. Next, we give few definitions, notations and assumptions referred in our subsequent discussions. Finally, we discuss the coverage criteria considered in our work.

UML 2.0 Sequence Diagrams

A sequence diagram precisely specifies the set of objects and the sequences of message exchanges that are involved in various scenarios. UML 2.x sequence diagram provides a mechanism known as combined fragments also known as interaction fragments. A combined fragment encloses one or more processing sequences in a frame which are executed under specific named circumstance called fragment operators. There is a facility for providing 12 different types of fragment operators. We briefly discuss only those interaction operators which are used in this work.

Combined fragment loop: A *loop* fragment indicates that the messages within the operand are to be repeated a number of times. The interaction constraint of a loop operand may include a lower and an upper limit specifying iterations of the loop as well as a Boolean expression. The loop fragment describes the test to be performed before the first execution of the messages in the loop operand indicating a pre-test form of loop. Since it is impractical to include all message paths of a loop fragment, we consider a pre-test form of loop criterion (defined in Section 3) to generate a test set based on selection of similar paths [16].

Combined fragment alt, opt and break: The fragments *alt, opt* and *break* denote a choice of behavior which is controlled by an interaction constraint. We denote this choice of behavior as selection and associate a selection criterion to include scenarios corresponding to each operand. The chosen operand has a constraint evaluated to true.

Combined fragment par: Typically, the inteaction fragment *par* denotes the parallel merge among the messages in the operands of a par fragment. The messages in a par fragment can be interleaved as long as the ordering imposed by each operand is maintained [6]. In this respect, we consider a valid interleaving sequence as the one which maintains ordering of message sequences within an operand.

UML Class Diagrams

For every class in the sequence diagram, the class diagram declares all the method signatures and class attributes. For a method m from a sender class C_1 to a receiver class C_2 in the sequence diagram, the class diagam defines the method signature m in class C_2. The method signature includes the name of the method, paramter type and return type whereas class attributes includes information about the instance variables such as their names and types. In addition, there may also be OCL constraints (Object Constraint Language) [27] which are used to express as invariants on the class attributes. These invariants specifies attribute constraints that are true for all instances of the class. This static information constrains the values of the attributes and hence included in the underlying class diagram as given below.

A class diagram is defined as $CD =< CL, AN >$ where CL is the finite set of classes present in the diagram and $AN =< C_1, AName, C_2 >$ is the set of association between classes in the diagram where $C_1, C_2 \in CL$ and $AName$ is the type of association. Each class $C_i \in CL$ is a tuple $C_i =< Attr, M >$ where $Attr$ is a set of class attributes $\{< attr_i : type_i, c_i >\}$. Each $attr_i$ is the name of the attribute with $type_i$ as the corresponding type of the attribute and c_i is the constraint over $attr_i$ specified as the Boolean expression. M is a set of method signatures $M = \{< m_i(p_1 : type_1, \cdots, p_n : type_n),\ Rtype_i >\}$ where m_i is the name of the method with parameters $p_1, \cdots, p_n$, associated types $type_1, \cdots, type_n$ and return type $Rtype_i$.

Structured Composite Graph (SCG)

In order to systematically investigate the comprehensive flow of control from a sequence diagram, the information contained in the sequence diagram is extracted and stored in a graph known as *structured composite graph*. The following nodes are considered while mapping a sequence diagram to a structured composite graph.

- An *initial* node represents the beginning of a structured composite graph.
- A *block* node represents a sequence of messages such as messages within operands of a fragment.
- A *decision* node represents a conditional expression such as Boolean expression that need to be satisfied for selection among operands of a fragment.
- A *merge* node represents an exit from the selection behavior such as an exit from an *alt* or an *opt* fragment.
- A *fork* node represents an entry into a *par* fragment.
- A *join* node represents an exit from a *par* fragment.
- A *final* node represents an exit of a scenario graph.

A Structured Composite Graph (SCG) is a directed representation of a sequence diagram. An SCG is defined as below.

A SCG $G =< A, E, in, F >$. Here, in denotes the initial node such that there is a path from in to all other nodes and F denotes a set of all final nodes representing terminal nodes of the graph. A is a set of nodes consisting of $(BN \cup CN)$ where BN is a set of block nodes, and $CN = (DN \cup MN \cup FN \cup JN)$ is a set of control nodes such that DN is a set of decision nodes, MN is a set of merge nodes, FN is a set of fork nodes and JN is a set of join nodes. Edges from decision nodes are labelled with a condition. E denotes a set of control edges such that $E = \{(x, y) | x, y \in A \cup F\}$. The structure of each node $A_i \in A$ is defined as given below.
$< nodeId, nodeType, nodeDetails, outEdge >$ where

- $nodeId$ is a unique label attached to each node.
- $nodeType = \{decision, merge, fork, join\}$ for each $C_i \in CN$ and $nodeType = \{block, initial, final\}$ for all other nodes.

- $nodeDetails = \{m_1, \cdots, m_q | q$ is a number of messages in $B_i \in BN$ $\}$. Each $m_i \in nodeDetails$ is defined as a triple $< m, s, r >$ with each message specifying its sender s, receiver r and name of the message m for all block nodes $B_i \in BN$. The sender and receiver denote classes and are in the form $< objectName, className >$. Each message m_i combines type information of the receiver class r from the class diagram and is structured as $< m, paramList, rValue >$. The type information is attached to both parameters $paramList = \{p_1, \cdots, p_n\}$ and return value $rValue$. Further, a parameter p_i of a method m_i in the sequence diagram may be a class attribute involved with constraint. A parameter or a return value derives both type and constraint information from the class diagram and is structured as $< name, type, value, constraint >$ where $name$ denotes the name of the parameter or the attribute with $type$ as their corresponding data type and $value$ is an instance of the value assigned. The $constraint$ refers to OCL expressions that are defined for the class attributes involved as method parameters.
  $nodeDetails = \{alt, loop, break, opt, par\}$ associates an interaction operator to a control node, $C_i \in CN$.

- $outEdge = \{OE_1, \cdots, OE_q \mid q$ is number of outgoing edges $\}$. Each $OE_i \in outEdge$ is defined as $< outNode, predicate >$ where $outNode$ specifies the $nodeId$ corresponding to the successor node and $predicate$ specifies the Boolean expression attached to an operand of a fragment.

For the automatic generation of test data, it is required to find test cases comprising sequences of method calls and test data. For this purpose, an aggregate model is to be defined combining behavioral and structural information. The structural information is needed to associate relevant range of values for each parameter and return value of the method calls appearing in the sequence diagram. In addition, it must also take into account, any attribute constraints which may limit the possible choices of data for each attribute. Hence, it is required to capture static information and augment with the information retrieved from a sequence diagram. SCG is an aggregate model combining information from class diagram and OCL constraints for facilitating test scenario generation and test data generation.

Terminologies

Variable: The term variable refers to parameters of method calls, return value of a method call or variables present in the interaction constraint of a sequence diagram. We distinguish two types of variables - basic types such as integer, float, boolean, character and composite types such as strings, object type which is collection of one or more basic types.
Domain of variables: The domain D_x of a variable x is a set of all values that the variable x may take. The domain is limited by the range of values allowed for a given variable in the computer. We denote the domain of variable x by

$D_x = (min(x), max(x))$ where $min(x)$ denote the lower bound and $max(x)$ denote the upperbound of admissible values for the variable x. We denote $X = \{x_1, \cdots, x_n\}$ as the set of all variables of a sequence diagram. The domain of the structured composite graph G is the product $D = D_{x1} \times D_{x2} \times \cdots \times D_{x_n}$ where D_{xi} is the domain for variable x_i. Note that a variable x_i may be of composite type in which case x_i contains individual elements $\{e_1, \cdots, e_p\}$ where e_i is of basic type. A single point x in the n-dimensional input space D is referred to as an *input* or *test input* [8].
Test case: A test case is a set of test inputs, preconditions, if any, for inputting the test data and expected output for a single execution of the program. A test suite is a collection of one or more test cases.
Scenario: A sequence $S =< n_0, n_1, ..., n_q >$ is a scenario or an abstract test case in a structured composite graph, if $n_0 = in, n_q \in F$ and $(n_i, n_j) \in E$, for all $i, j, 0 \leq i, j < q$, where E denotes a set of edges and F denotes a set of final nodes in a scenario graph.
Feasible Scenario: A scenario is feasible (or executable) if there exists an input X in the n-dimensional input space $X \in D$ for which the path is traversed during the program execution; otherwise, the path is infeasible (or unexecutable).
Constraint and Predicates: A constraint is a pair of algebraic expressions related by one of the relational operators $\{>, \geq, <, \leq, =, \neq\}$. An algebraic expression is composed of variables, parentheses, constants and arithmetic opeators such as $\{+, -, *, /\}$. A clause is a list of constraints connected by the logical operators {AND, OR }. A predicate is a list of clauses associated to a decision node. The predicates on the scenario arise from the guard expression of fragments such as *opt,loop, alt and break* and appear on the branches emerging from decision nodes. Note that a path can be represented by a list of predicates with one predicate for each decision node.

Coverage Criteria

In this subsection, we list the coverage criteria that are used in this work. Let T be a set of test cases for a sequence diagram. T satisfies the coverage criterion if the following condition holds good.

C1: All message path criterion

- For each sequence diagram in the model, T must include test cases to execute all message sequence paths of the sequence diagram [28].

C2: Loop adequacy criterion
For each loop fragment,

- T must include at least one scenario in which control reaches the loop and then the body of the loop is not executed ("zero iteration" path).
- T must include at least one scenario in which control reaches the loop and

then the body of the loop is executed at least once before control leaves the loop ("more than zero iteration" path).

C_3 : *Selection coverage criterion*

- For each selection fragment, T must include one scenario corresponding to each evaluation of the constraint.

C4: Concurrent coverage criterion

- For each concurrent node in SCG, T must include one scenario corresponding to every valid interleaving of message sequences.

4 PROPOSED METHODOLOGY

Scenarios derived from the sequence diagrams describe the functionality of a system under development in terms of its behavioral descriptions. In the context of system testing, scenarios representing an abstract level of test cases need to be augmented with test data. In this section, we present our approach for generating test cases from a sequence diagram.

Building SCG

In this sub section, we discuss our approach of building a SCG for test case generation. As defined in Section 3, a SCG is a directed graph which is obtained by integrating the necessary information from a class diagram, OCL constraints and a sequence diagram.

The sequence diagram models the interaction among a set of objects using sequences of messages and interaction fragments. An important task in the test case generation therefore, is to extract the flow of control among the fragments and their nested occurrences. The idea of SCG is to express the underlying control flow information involved in a sequence diagram as a directed graph. Two types of nodes are considered in SCG. Similar to the notion of block of program instructions [7], a *block node* in SCG is a node corresponding to a set of messages from the sequence diagram. Since a fragment is expected to alter the flow of control, a *control node* is used to mark the entering and leaving of a fragment. Depending on the type of interaction operator, four types of control nodes are used. An interaction operator of type *alt, loop, break* and *opt* denote conditions that enable the selection of interaction operands. Accordingly, a decision node is used for displaying the selection behavior. An edge from the decision node is associated with a conditional expression that enable the selection of an interaction operand. A merge node is used for displaying exit from the selection behavior. A set of fork and join node is used as an entry and exit from a par fragment.

The parameters of a method call in the sequence diagram lacks the constraint and type information. This additional information is derived from the class diagram and is appended to each message. Hence the structure of each variable v in the block node is $(name, type, value, constraint)$. Here, *type* and *constraint* refer to the data type and the attribute constraints associated to a variable $v.name$. The *type* information is used to map each variable $v.name$ to a range of values (min, max). This ensures that each variable on a path from the initial node to a final node is mapped to a range of values. Furthermore, the *constraint* information is used to appropriately set the boundary values min and max. For example, the type information such as *int* to a variable x sets $(min_x, max_x) = (minInt, maxInt)$ where $minInt$ and $maxInt$ are the boundary values. Let us consider $x \geq 5$ as the constraint associated to x. In that case, (min_x, max_x) will be set to $(5, maxInt)$ which will be recognized as the initial domain.

Approach to build the SCG from a given sequence diagram is stated in the algorithm *CreateSCG*. It is defined in terms of a function *exitNode = ProcessCompositeInfo(fragmentId, classDiagramId, entryNode)* to extract the control flow within nested fragments. Initially, the main frame that hosts the sequence diagram of a use case is supplied as a *fragmentId*. Depending on the contained elements within this main frame, the function is recursively called with the *fragmentId* of the element to be transformed. When the element derived from the sequence diagram is, say message m, then the receiver class of the message is consulted. The method signature corresponding to the method call is then derived using the function *ReturnMessageStructure*. For the OCL constraints, we assume that, attribute constraints, if any, are available in the attribute structure as mentioned in Section 3 (B). Both constraint and type are then appended to the message m. The code to perform this is stated in the function call *AttachTypeInfo()* and *AttachConstraintInfo()*. In addition, with each element of the sequence diagram, we distinguish two nodes- entry node and exit node. The entry node is the current node which is connected to the outside by incoming edges and therefore supplied as input to the function. The exit node is the node which is connected to the outside by outgoing edges and hence returned as output of the function. The fragments are processed until the termination condition is reached. When the termination condition for the main fragment is reached, the scenario graph is returned with initial node in as the entry node and the final node fn as the exit node. In this way, SCG is built from a sequence diagram with any nested combination of fragments.

The structure of SCG corresponding to each of the interaction fragment is shown in Figure 1. The corresponding class diagram is consulted in each case to append static information. In SCG, block nodes are shown in ovals and only node-id is mentioned for each of the nodes, for brevity. The guard associated to a fragment is shown as an edge descriptor. For denoting fork and join nodes thick line segments are considered whereas for denoting decision and merge nodes diamond symbols are used. A filled circle notation is used for denoting initial node and a small circle

Procedure 1 Function CreateSCG

```
Input: $D$: Sequence diagram in XMI form          // $D$ is the main fragment
    $CD$: Class diagram in XMI form
Output: $G$: SCG in the form $< A, E, in, F >$
 1: Create initial node $in$ ;
 2: $x = ProcessCompositeInfo(D, CD, in)$ ;       // Process the sequence diagram $D$
 3: if $x \neq finalNode$ then
 4:     Create final node $fn \in F$ ;
 5:     Connect edge from $x$ to $fn$ ;
 6: end if
 7: return $G$ with entry node $in \in A$ and exit node $fn \in F$ ;
 8: stop
```

Algorithm - The SCG building algorithm

enclosing an 'X' is shown for denoting final nodes. Further, node label B_i is used for denoting block nodes. FN_i and JN_i labels refer to fork and join nodes, respectively. For denoting decision and merge nodes D_i and M_i are used as node labels.

Generating Scenarios

Input to the scenario generation procedure is SCG. The output from the scenario generation procedure is a finite set of scenarios which are complete paths starting from the initial node to a final node.

Each scenario is built starting with the initial node of the SCG. The sequence diagram based coverage criteria (defined in Section 3) are used to generate a set of scenarios which are to be covered during testing. These scenarios ensure that the elements modeled in the sequence diagram are traversed at least once.

All variables in the block node are associated with $(type, constraint)$ information which is used to set initial domain. One representative value for each variable on the path is to be selected. Hence, messages in block nodes along a path correspond to a parameterized operation call. Each outgoing edge from a decision node contains one predicate. A test case must satisfy all predicates along its path. Traversing nodes and edges of SCG to find test paths therefore correspond to depth first search traversal. The main algorithm is shown in *GenerateTestScenario*. It uses depth first search technique on the SCG to find a test scenario.

Synthesizing Test Data

The test scenarios obtained as discussed in the previous section denote abstract test cases which represent possible traces of executions. Consequently, the sequence of messages comprising a scenario is a feasible sequence of messages. To generate test data for a scenario is to find *test input* that satisfies all the constraints along the

Procedure 2 Function ProcessCompositeInfo

Function ProcessCompositeInfo(Fragment : fragmentId, CD : class Diagram Id, A : curNode)

```
Input: $fragmentId$: Fragment, a tag indicating the type of fragment
       $classDiagramId$ : Id indicating the name of the class diagram
       $curNode \in A$
Output: $exitNode \in A$
 1: while ! EndOfFragment do // end of current fragment
 2:   $x = GetNextElement()$ ;          // Read the next element in the fragment
 3:   if $x =' EOF'$ then // Termination condition
 4:     $exitNode = curNode$ ;
 5:   else if $x =' message'$ then
 6:       begin
 7:        $BN = CreateBlockNode()$          // Create $BN$ with a set of messages ;
 8:         for $each\ message\ m \in B$
 9:         begin
10:           Get the receiver class in $r.className$
11:           Msg = ReturnMessageStructure(CD, r.className, m) // Retrieve method

12:           Attr = ReturnAttributeStructure(CD, r.className) // Return attribute
13:           for all variables in $m$
14:           begin
15:             AttachTypeInfo(Msg,m) // Construct message structure
16:             AttachAttributeInfo(Attr,m) // Attach constraint $c[i]$ to $Msg.p[i]$
17:           endfor
18:         endfor
19:         ConnectEdge($curNode, BN$)          // Edge from $curNode$ to the next node

20:         $exitNode = BN$ ;
21:   end if
22:   return $exitNode$
23: end while
```

Function ProcessCompositeInfo(D, CD, A)

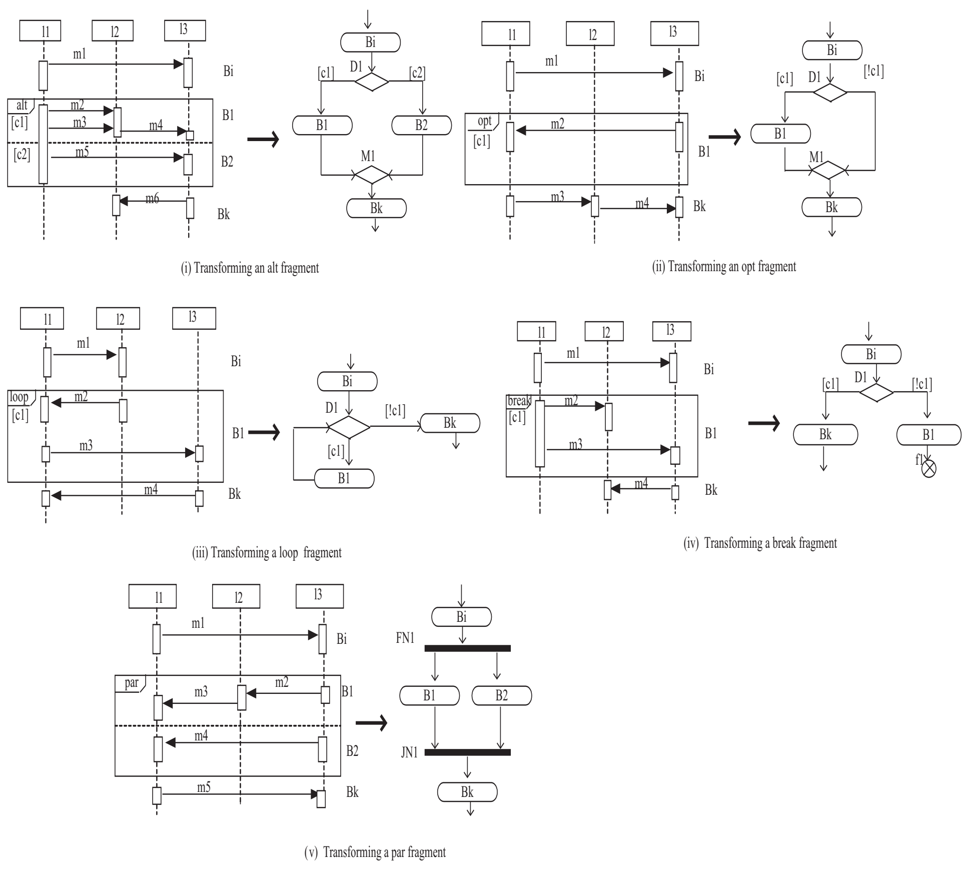

l1
l2
l3
m1
Bi
alt
[c1]
[c2]
m2
m3
m4
m5
m6
B1
B2
Bk
D1
M1
(i) Transforming an alt fragment
opt
[!c1]
(ii) Transforming an opt fragment
loop
(iii) Transforming a loop fragment
break
f1
(iv) Transforming a break fragment
par
FN1
JN1
(v) Transforming a par fragment

Figure 1: General structure of SCG

Procedure 3 Function GenerateTestScenario

Input: SCG, a graph with initial node in .

Output: T, a collection of test cases forming the test suite.

```
Data: current: The next node being visited
1: current = in
2: while current ≠ NULL do
3:     begin
4:   push(current)
5:   if current = finalNode then
6:     updatePath(T)
7:   end if
8: end while
```

Algorithm - Test scenario generation from SCG path.

We consider a domain based test data derivation for generating test input. Our aim is to derive a feasible domain for each variable of the sequence diagram such that a value from the domain can cause the execution of the scenario. Let $ID_{xi} = (min_{xi}, max_{xi})$ be the initial domain (ID) assigned to a variable x_i. It is required to find a feasible domain $FD_{xi} = (bot_{xi}, top_{xi})$ which satisfies all the constraints along the path where, $bot_{xi} \geq min_{xi}$, $top_{xi} \leq max_{xi}$ and $bot_{xi} \leq top_{xi}$. If such a domain cannot be found, then the path remains infeasible. Note that, for test data generation, it is not required to find all possible data values. Therefore, our aim is to find any sub domain of the feasible domain so that the current scenario can be executed. In this context, we state our test data synthesis problem as given below.

For a test scenario t_i denoted as a sequence of nodes $< n_{i1}, n_{i2}, \cdots, n_{iq} >$ where n_{i1} denotes the initial node and n_{iq} denotes the final node, we need to find a test input satisfying all the constraints along the path such that the path reaches the final node n_{iq}.The synthesis procedure includes the following steps:

Scenario Interpretation

In the path based test data generation approaches, each of the variable appearing in the predicate is expressed in terms of input variables. The symbolic values are assigned in symbolic execution approaches whereas actual execution values are assigned in case of dynamic test data generation approaches. Unlike program testing approaches, the values of variables cannot be traced while generating system test cases from model based specifications. The variable on the predicates must be appeared either as a parameter or a return variable in a method call. This interpretation associates every variable involved in the predicate to its domain. For example, a variable in the predicate defined as *integer* in the class diagram is allowed to take only integer values. A test scenario is well defined if the variables in the predicates are declared in the class diagram. Hence, to facilitate predicate interpretation, the sequence diagram is assumed to be well defined which is a pre-requisite for test data generation procedure.

The scenario interpretation is carried out from the initial node to a final node of the selected scenario. The given scenario is analyzed looking for method calls and predicates which in turn, are handled as explained in the following two steps:

Step 1: Deriving Initial Domains
For each method call of a block node, the initial domain is associated to its parameters and return variables. The initial domain is associated with a variable in one of the following ways.
1) The initial domain can be assigned by *type* information which is attached to SCG with every method call. For example, a type information such as *integer* associated to a variable x, assigns a domain (MIN_x, MAX_x) where MIN_x and MAX_x denote the allowable lower bound and upper bound of values for an integer variable.

2) For each variable, the initial domain which is the domain of possible values is constrained by attribute constraints. These constraints extracted from OCL constraints are attached to SCG with parameters of the method call. For example, a variable x with OCL constraint $x > 100$ constrains the initial domain of variable x as $(1, MAX_x)$ where lower bound value $min = 1$ and upper bound value $max = MAX_x$.

A data structure known as *symbol table* is built to record domain information. The symbol table stores the names and types of the variables and their associated domains corresponding to every test case. Since all the variables are entered into the symbol table, the symbol table is an important data structure. There exists an entry in the symbol table for every variable encountered in the predicate. In other words, the symbol table ensures that a scenario is well defined before solving the predicates.

Table 1 shows the structure of a symbol table. The name of the variable is the parameter name associated with a method call on the sequence diagram. Each variable is given a data type containing one of the following types - basic type such as integer, float, boolean, character or composite type such as strings, objects. The initial domain is determined according to the data type of the variable. After solving all the predicates along the path, the final domain is generated.

Table 1: Symbol Table Structure

Variable	Data Type	Initial Domain	Final Domain
x	int	(minInt, maxInt)	(100, maxInt)
y	string	"SET, OPEN, CLOSE"	"OPEN, CLOSE"
...	...	...	...

Step 2: Constructing Expression Trees
Each predicate of a decision node is represented internally as an expression tree. The structure of expression tree is similar to those used in compilers [29] while parsing expressions. The terminal nodes (leaves) of an expression tree are the variables or constants in the expression. The non-terminal nodes of an expression tree are the operators such as arithmetic operators (+, -, *, /), relational operators, boolean operators or logical operators. Each predicate along the path is separately represented as an expression tree. Considering the hierarchy of operators in the constraint, the tree structure is built to utilize this hierarchical order. We consider predicates with following forms:

$$\begin{array}{ll}
Predicate & ::= Clause_1\ B_{op}\ Clause_2\ B_{op} \cdots\ B_{op}\ Clause_n \\
Clause & ::= LeftConstraint\ R_{op}\ RightConstraint \\
LeftConstraint & ::= Expr \\
RightConstraint & ::= Expr \mid Const \\
Expr & ::= Expr\ A_{op}\ Expr \mid Expr\ A_{op}\ Var \mid Expr\ A_{op}\ const \mid Var \\
A_{op} & ::= + \mid - \mid * \mid / \\
R_{op} & ::=> \mid \geq \mid < \mid \leq \mid = \mid \neq \\
B_{op} & ::= AND|OR
\end{array}$$

Let us consider an example to see how expression trees can be constructed. Consider a scenario described using the predicate $x \geq 100\ AND\ y \neq set$. After entering the variables into the symbol table, the expression tree for the predicate constructed is shown in Figure 2. However, a test case needs to satisfy all predicates along its path. If the evaluation of the next predicate on the path is inconsistent with the existing set of test values (or their domain), then the path is infeasible. This requires that every predicate be checked to determine whether it satisfies the previous solution. In the proposed expression tree, the variables appear as leaves and for each evaluation, the variables receive a new domain. This value is then subsequently propagated till it reaches the root for evaluation (constraint solving is explained in the next sub section). The tree structure helps in propagating values while simplifying expressions, which in turn, helps in simplifying path evaluation.

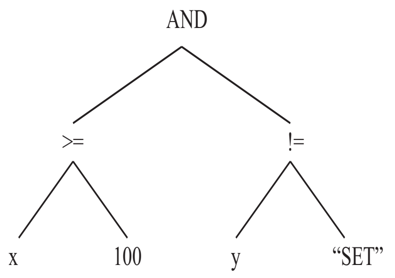


Figure 2: Structure of an Expression tree

Constraint Solving

In this step, we find a feasible domain for each variable in the sequence diagram using dynamic domain reduction procedure [30]. The domain reduction procedure tries to satisfy each predicate along the scenario by reducing the initial domains for the variables involved in the predicate. The reduced domain at the end of the procedure denotes the final (feasible) domain, any value from this domain will cause the execution of the path.

Given the initial domains of two variables known as left and right variables that are combined by a relational expression, if the initial domains are non-intersecting, then the clause may be either satisfied or is infeasible. On the other hand, if the domains are intersecting, then the reduction procedure is carried out to split the domains such that the clause is satisfied for all values from the two domains. Let us assume that the initial domains of left variable be $(left.bot, left.top)$ and right variable be $(right.bot, right.top)$. The split point is found based on top and bottom values of left and right domains. There are four cases to consider.

(a) $(left.bot \geq right.bot)$ and $(left.top \leq right.top)$
$split = (left.top - left.bot) * pt + left.bot$

(b) $(left.bot \leq right.bot)$ and $(left.top \geq right.top)$
$split = (right.top - right.bot) * pt + right.bot$

(c) $(left.bot \geq right.bot)$ and $(left.top \geq right.top)$
$split = (left.top - right.bot) * pt + right.bot$

(d) $(left.bot \leq right.bot)$ and $(left.top \leq right.top)$
$split = (right.top - left.bot) * pt + left.bot$

Once a split point is returned, the domains of left and right variables are adjusted. Table 2 shows the reduced domains of left and right variables. In this table, $offset$ is the constant value and is included to hold the smallest possible difference between two values of a given data type. For integer and floating point values offsets are taken as 1 and 0.00001, respectively. Based on the precision and accuracy to be maintained, any other suitable offset for floating point values can be chosen. Thus $offset$ helps in setting domains based on the type definition of a variable. The value $split$ is the split-point computed as explained above. For relational operator, "=", the domain of left and right variable has to be chosen in such a way that the value chosen should be the same for both left and right variables. This is indicated in Table 2 by choosing the reduced domains as $(split, split)$ for both left and right variables.

The value of pt is initially set to 1/2 to get the new domains. The leaves of the expression tree are inspected for computing domains. If the leaves denote variables, then the symbol table is consulted to get an initial domain. Substituting variables with their initial domains and choosing split point $pt = 1/2$, the new domains are computed. If the new domains satisfy the predicate then the procedure continues with the next node of the scenario until the final node is reached. If the new domains do not satisfy the clause then the value of pt is changed to 1/4 and a different split is found for the same decision node. The value of pt is chosen sequentially from the set $(1/2, 1/4, 3/4, 1/8, \cdots)$. The procedure of splitting domains to get new domains is repeated for a fixed number of iterations, each time by choosing a next search point *pt* from the above set. If the iterations fail to satisfy the predicate, then the procedure is repeated from the previous decision node. If the procedure again fails with the previous node and if there are no more decision nodes to consider, then the test data generation fails and a new scenario is taken. The reduced domain at the end of the procedure denotes a feasible domain of values for a test case.

Table 2: Determining reduced domain based on the type of relational operator

Relational operator	*Domain of left variable*	*Domain of right variable*
$\geq$	$(split, left.top)$	$(right.bot, split)$
$>$	$(split + offset, left.top)$	$(right.bot, split)$
$\leq$	$(left.bot, split)$	$(split, right.top)$
$<$	$(left.bot, split - offset)$	$(split, right.top)$
$=$	$(split, split)$	$(split, split)$
$\neq$	$(left.bot, split)$	$(split + offset, right.top)$

A clause of a predicate may get reduced in one of two ways.
(a) When a clause is of the form $Var\ R_{op}\ const$, it reduces the domain of values for Var. Let us consider a clause $x \geq 100$ with $ID_x = (1, 200)$. After domain reduction procedure, the domain of x will be FD_x = (100, 200).
(b) When a clause is of the form $Var_1\ R_{op}\ Var_2$ then the domain of values for both Var_1 and Var_2 are reduced. Let us consider a clause $x \geq y$, which reduces the domain of x and y. Let the initial domain of x and y be $ID_x = (1, 20)$ and ID_y = (1, 30). In that case, the feasible domain for x and y after domain reduction procedure will be $FD_x = (11, 20)$ and $FD_y = (1, 10)$. In this computation, the split point 10 is obtained whereas offset 1 is chosen considering integer domain. Thus, final domain of x is assigned as (11, 20) as per relational operator ” $>$ ” of Table 2.

5 ILLUSTRATION OF THE PROPOSED APPROACH

In this section, we illustrate our test data synthesis approach using an example called *Trading House Automation (THA) system*. We consider a sequence diagram for *orderItem* use case of THA system as shown in Figure 3(a). The sequence diagram depicts the sequence of operations when the user, namely *inventoryManager* places order for the required products. The sequence diagram shows the interaction between actor with four system objects which are *:Controller, :CategoryManager, :ProductInfo* and *:OrderHandler*. The messages in the sequence diagram focus on synchronous communication between objects. For each object used in the sequence diagram, the class diagram shown in Figure 4 gives the classes and their relationships. For the sake of brevity, the class diagram shows only the specific classes that are involved in *orderItem* use case where the remaining classes and their relationships are not shown.

Table 3: Test Scenarios

Scenario Id	Test Scenario
T_1	$in, B1, D1, B2, D2, B3, M1, M3, B8, fn$
T_2	$in, B1, D1, B2, D2, B4, M1, M3, B8, fn$
T_3	$in, B1, D1, B5, D3, B6, M2, M3, B8, fn$
T_4	$in, B1, D1, B5, D3, B7, M2, M3, B8, fn$

SCG Construction

As part of SCG construction algorithm, the messages are enclosed in block nodes. Initially, the messages m_1 and m_2 are enclosed in a block node, which is named as B_1. In the block node, the information pertaining to the method data type and constraints from the class diagram are recorded as part of message structure stated in Section 3. The next element retrieved from the sequence diagram is the *alt* fragment. A decision node named $D1$ is created with two outgoing edges corresponding to two operands of the *alt* fragment. Each interaction operand contains an inner fragment, corresponding to which fragments are processed accordingly by creating control nodes as discussed in Section 4. The block nodes after processing the sequence diagram are marked sequentially on the right side of Figure 3(a). The structured composite graph (SCG) is shown in Figure 3(b).

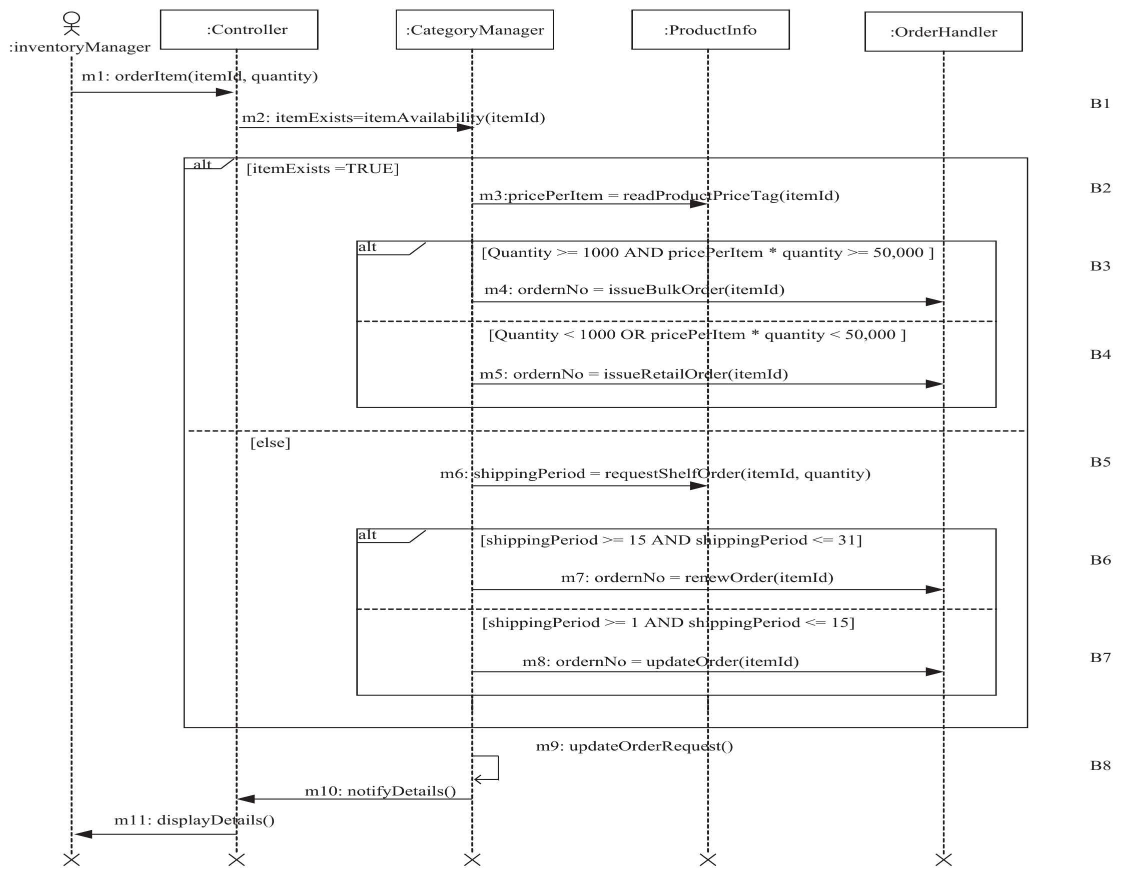
Scenario Generation

To generate scenarios, the SCG model is traversed in a depth first order manner. The control nodes that detect beginning and ending of a fragment are identified during traversal. For the set of nodes lying within a fragment, the paths are then generated to satisfy the respective coverage criterion. For example, for a loop fragment, two representative paths are formed from the loop entry node to the loop exit node to cover the loop adequacy criterion. In our example sequence diagram shown in Figure 3(a), the scenarios have to be generated from a nested alt structure. With respect to each decision node, the sub paths are therefore generated to cover all outgoing edges. As a result, four scenarios are generated from the SCG shown in Figure 3(b). The test specifications are listed in Table 3.

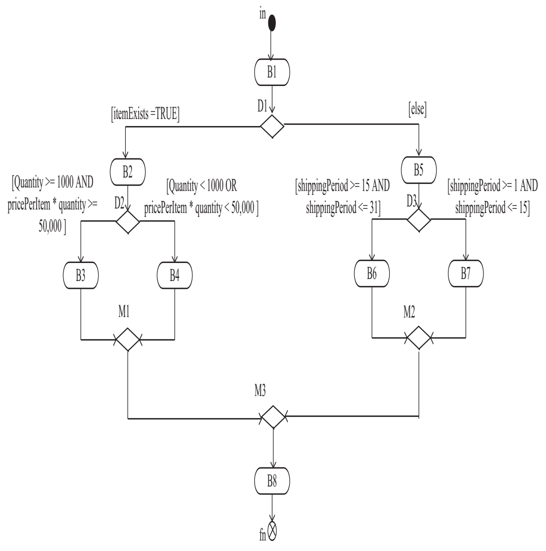
Test Data Synthesis

Deriving Initial Domain

Consider the scenario described by $T1 =< in, B1, D1, B2, D2, B3, M1, M3, B8, fn >$. In the first step, all variables receive their initial domain of values. The symbol table built during scenario interpretation step maintains this information. Starting from the initial node, each of the variable along the scenario is entered into the


:inventoryManager
:Controller
:CategoryManager
:ProductInfo
:OrderHandler
m1: orderItem(itemId, quantity)
B1
m2: itemExists=itemAvailability(itemId)
alt
[itemExists =TRUE]
B2
m3:pricePerItem = readProductPriceTag(itemId)
alt
[Quantity >= 1000 AND pricePerItem * quantity >= 50,000]
B3
m4: ordernNo = issueBulkOrder(itemId)
[Quantity < 1000 OR pricePerItem * quantity < 50,000]
B4
m5: ordernNo = issueRetailOrder(itemId)
[else]
B5
m6: shippingPeriod = requestShelfOrder(itemId, quantity)
alt
[shippingPeriod >= 15 AND shippingPeriod <= 31]
B6
m7: ordernNo = renewOrder(itemId)
[shippingPeriod >= 1 AND shippingPeriod <= 15]
B7
m8: ordernNo = updateOrder(itemId)
m9: updateOrderRequest()
B8
m10: notifyDetails()
m11: displayDetails()

(a) The sequence diagram for *orderItem* use case of Trading House Automation (THA) system


in
B1
D1
[itemExists =TRUE]
[else]
B2
B5
[Quantity >= 1000 AND pricePerItem * quantity >= 50,000]
D2
[Quantity < 1000 OR pricePerItem * quantity < 50,000]
[shippingPeriod >= 15 AND shippingPeriod <= 31]
D3
[shippingPeriod >= 1 AND shippingPeriod <= 15]
B3
B4
B6
B7
M1
M2
M3
B8
fin

(b) The SCG of the sequence diagram

Figure 3: Illustrating SCG creation algorithm

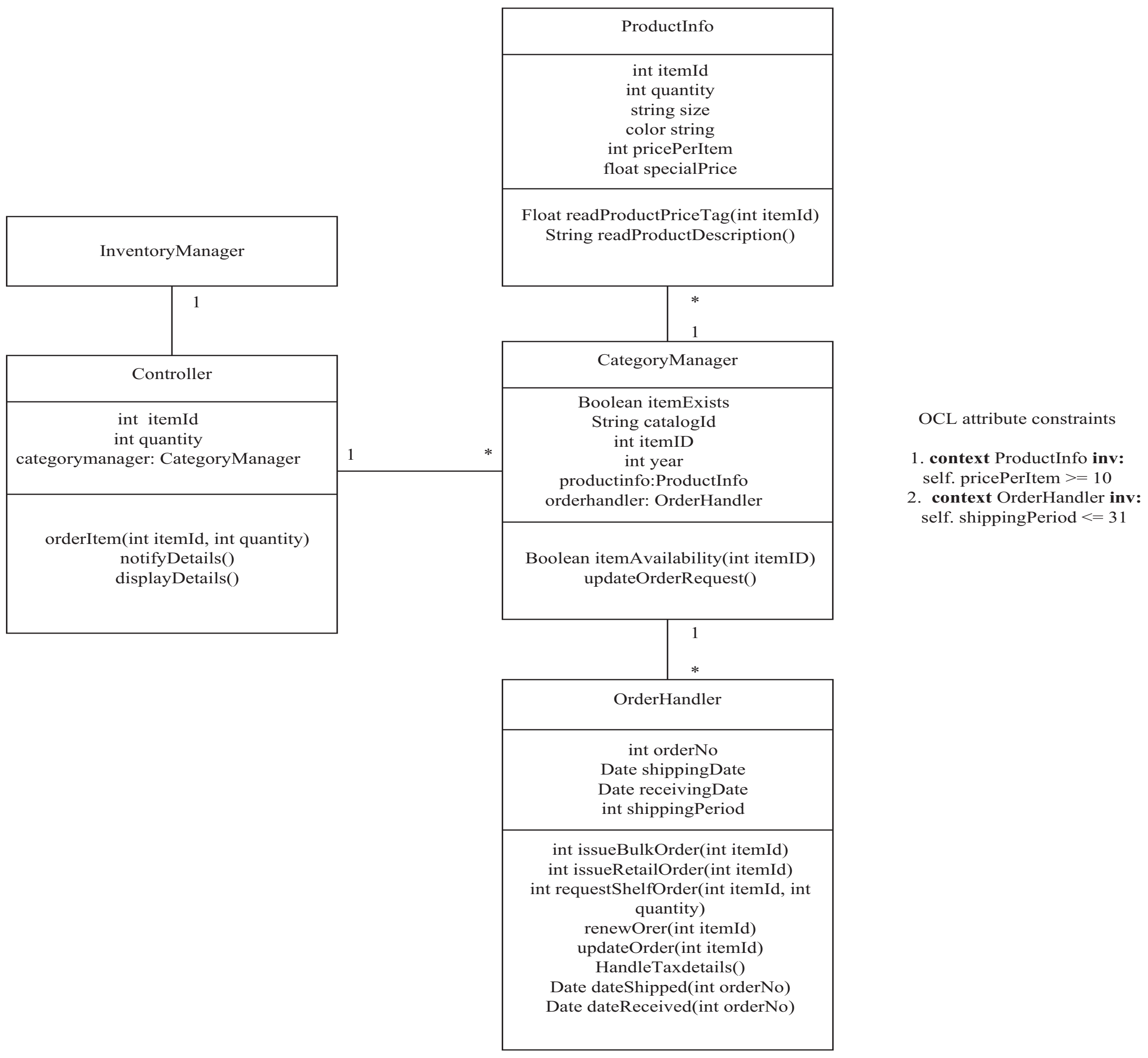

ProductInfo
int itemId
int quantity
string size
color string
int pricePerItem
float specialPrice
Float readProductPriceTag(int itemId)
String readProductDescription()
InventoryManager
1
*
1
Controller
int itemId
int quantity
categorymanager: CategoryManager
orderItem(int itemId, int quantity)
notifyDetails()
displayDetails()
1
*
CategoryManager
Boolean itemExists
String catalogId
int itemID
int year
productinfo:ProductInfo
orderhandler: OrderHandler
Boolean itemAvailability(int itemID)
updateOrderRequest()
1
*
OrderHandler
int orderNo
Date shippingDate
Date receivingDate
int shippingPeriod
int issueBulkOrder(int itemId)
int issueRetailOrder(int itemId)
int requestShelfOrder(int itemId, int quantity)
renewOrer(int itemId)
updateOrder(int itemId)
HandleTaxdetails()
Date dateShipped(int orderNo)
Date dateReceived(int orderNo)
OCL attribute constraints
1. context ProductInfo inv: self. pricePerItem >= 10
2. context OrderHandler inv: self. shippingPeriod <= 31

Figure 4: UML class diagram of Trading House Automation (THA) system

symbol table (Table 4). The variables $itemId$ and $quantity$ of class $Controller$ are set with integer domain. Next, $itemExists$ variable of message $m2$ is entered into the symbol table. The next node on the path is decision node $D1$ from which the constraint $itemexists = TRUE$ is derived. At this time, the symbol table is checked to see that all variables in this predicate have an entry into the symbol table. Since the variable $itemexists$ is already in the table (see Table 4), the next node is retrieved as part of scenario interpretation step. If at any time, a variable involved in the predicate is not found in the symbol table, then the interpretation of the current scenario is terminated and the next scenario from the abstract test suite is chosen. Following the scenario, the next node in $T1$ is block node $B2$ with a single message $m3$. Connected to message $m3$, there exists a variable $pricePerItem$ with a constraint $pricePerItem \geq 10$. This constraint sets the initial domain of variable $pricePerItem$ as (10, maxInt) in the symbol table. The decision node $D2$ is now encountered giving rise to the constraint $quantity \geq 1000$ AND $pricePerItem * quantity \geq 50000$. Again, the variables in the predicate are checked in the symbol table to find an entry for each. The block node $B3$ is now inspected and the variables are entered in the similar manner as explained above. The node $B3$ is followed consecutively by two control nodes $M1$ and $M3$. These control nodes mark the begin and end of a fragment and are used only while generating paths corresponding to a coverage criterion. For the test data synthesis phase, these nodes are ignored and the node $B8$ is now investigated. The messages $m9, m10$ and $m11$ are examined. As there are no parameters, the scenario interpretation is terminated on encountering the final node fn as the next node.

Constraint Solving

The scenario interpretation step ensures that the variables are defined in the symbol table, which in turn ensures that the scenario is well defined. The collected constraints at the end of interpretation step is solved for finding test inputs. There are two constraints in scenario $T1$: $itemExists = TRUE$ and $quantity \geq 1000$ AND $pricePerItem * quantity \geq 50000$. In order to solve these constraints, the predicates are expressed in terms of expression trees. The domain reduction procedure is illustrated with each of these predicates. The first predicate $itemExists = TRUE$ is in the form $Var\ R_{op}\ const$. For this expression, the left variable $itemexists$ is related to the right variable using relational operator =. Since the right variable is a constant expression, the value after solving the constant is assigned to $(left.bot, left.top)$ as the domain of left variable. Hence the final domain of variable $itemexists$ is set to $(TRUE, TRUE)$.

The expression tree for the second predicate is shown Figure 5. The constraint solving for this expression begins evaluation using the initial domains of variables. The left variable is a constraint $quantity \geq 1000$. Hence, the expression is in turn decomposed recursively into the left variable $quantity$ and the right variable 1000. For the constraint in the form $Var\ R_{op}\ const$, the final domain of $quantity$ is set to $(1000, maxInt)$. Now the expression $pricePerItem * quantity \geq 50000$ is assigned as the right variable and the constraint solving is continued.

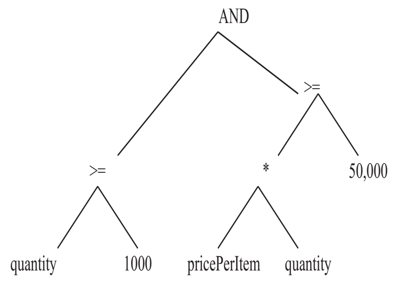


Figure 5: Expression tree for the predicate $quantity \geq 1000\ AND\ pricePerItem * quantity \geq 50000$

For an expression $pricePerItem * quantity$, and a value 50000, it is required to set domains for individual variables $pricePerItem$ and $quantity$ such that the expression has the required value. The expression evaluation is done in two phases. In the first phase, the domain of the expression is computed based on the domains of individual variables. In the second phase, based on the right hand side of the expression, the domain of individual variables are re-adjusted by propagating changes down the leaves. This necessitates forward traversal of expression tree which is from the leaves to the root followed by back ward traversal which is from the root to the leaves. To do forward traversal, the initial domain is retrieved from the symbol table which is $pricePerItem : (10, maxInt)$ and $quantity : (1000, maxInt)$. In order to evaluate expression, a suitable *bottom* value is chosen and the domain is readjusted accordingly. Let the new domains be $pricePerItem : (10, 100)$ and $quantity : (1000, 1500)$. Based on these domains, the domain of the expression $pricePerItem * quantity$ is computed as $top = max((left.bot * right.bot), (left.bot, right.top), (left.top*right.bot), (left.top*right.top)$ and $bottom = min((left.bot*right.bot), (left.bot, right.top), (left.top*right.bot), (left.top*right.top)$. This leads to $(10, 000, 1, 50, 000)$ as the domain of expression. The expression is now in the form $Var\ R_{op}\ const$ with $(10, 000, 1, 50, 000)$ as the domain of Var. Based on the $const$ value, which is 50,000, the domain of expression is now readjusted as $(50, 000, maxInt)$. After choosing a suitable bottom value, the domain of expression becomes $(50, 000, 55, 000)$. The changes are to be propagated to the leaves by backward traversal. The adjusting of domain is done by taking an inverse operation [30]. For the arithmetic operator $*$, the inverse operation is computed as $left, right : (SQRT(bot), SQRT(top))$ which is $pricePerItem : (224, 234)$ and $quantity : (1000, 1500)$. The symbol table for T_1 is shown in Table 4. Similarly, scenario T_2 is computed by taking initial domain for $pricePerItem$ as $(10, 100)$ and $quantity$ as $(1, 1000)$. The symbol table with the initial and final domain is shown in Table 4 for all four scenarios. The variables which are not involved in the constraints are unchanged and are not shown in this table.

Table 4: Symbol Table for *orderItem* use case

ScenarioId	Variable	Data Type	Initial Domain	Final Domain
T_1	itemExists	Boolean	(FALSE, TRUE)	(TRUE,TRUE)
	quantity	integer	(minInt, maxInt)	(1000, 1500)
	pricePerItem	integer	(10, maxInt)	(224, 234)
T_2	itemExists	Boolean	(FALSE, TRUE)	(TRUE,TRUE)
	quantity	integer	(minInt, maxInt)	(3, 223)
	pricePerItem	integer	(10, maxInt)	(10, 223)
T_3	itemExists	Boolean	(FALSE, TRUE)	(FALSE,FALSE)
	shippingPeriod	integer	(minInt, 31)	(15, 31)
T_4	itemExists	Boolean	(FALSE, TRUE)	(FALSE,FALSE)
	shippingPeriod	integer	(minInt, 31)	(1, 15)

6 CONCLUSIONS

In this paper, an automatic approach to test data synthesis is presented. The test effectiveness of the system depends on the selection of test cases. In this regard, selecting test data and identifying test data boundary is an important task. The OCL expressions of system models are considered in this approach for constraining the boundaries of variables and for assigning initial domain to the variables involved in the scenario. Since this is done automatically from the OCL expressions, the approach derives effective test data for all the variables of a sequence diagram. The result of test data synthesis denotes a feasible domain which is a sub-domain of the initial domain for the selected scenario.

A challenging problem with path-oriented test data generation is the determination of infeasible paths which means that there is no input data for them to be executed. Existing model based test data generation approaches ([20], [22], [23], [24], [25], [26]) including the proposed approach assume that initially all test paths are feasible. If a path cannot be exercised by any set of input data, then the path becomes infeasible. An infeasible path exists because contradictory constraints are required to be satisfied to execute the path. A significant effort which is wasted in trying to satisfy inconsistent constraints can be saved by detecting the infeasible scenarios. It is required to extend the approach for detecting contradictory constraints based on domain information.

REFERENCES

[1] Robert V. Binder. *Testing Object Oriented Systems: Models, Patterns and Tools*. The Addison-Wesley Object Technology Series, 1999.

[2] L. Briand and Y. Labiche. A UML- based approach to system testing. *Journal of Software and Systems Modeling*, pages 10–42, 2002.

[3] Junit home page. http://www.junit.org/.

[4] Jsystem home page. http://www.jsystemtest.org/.

[5] Jwebunit home page. http://jwebunit.sourceforge.net/.

[6] UML. *UML 2.0 Superstructure - Final Adopted Specification.* Object Management Group, 2003. http://www.omg.org/docs/ad/03-08-02.pdf.

[7] Frances E. Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, July 1970.

[8] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[9] B. Korel. Automated test data generation for programs with procedures. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 209–215, 1996.

[10] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085 – 1110, December 2001.

[11] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[12] W. E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, 1977.

[13] J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1:391–409, 1991.

[14] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[15] Boris Beizer. *Software Testing Techniques.* International Thomson Computer Press, New York: Van Nostrand Reinhold, 2 edition, 1990.

[16] William E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, c-24(5):92–96, 1975.

[17] Simeon Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, June 1988.

[18] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–84, April 1983.

[19] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[20] Orest Pilskalns, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert France. Testing UML designs. *Information and Software Technology*, 49(8):892912, August 2007.

[21] Bingchiang Jeng. Toward an integration of data flow and domain testing. *Journal of Systems and Software*, 45(1):19–30, 1999.

[22] T. Dinh-Trong, S. Ghosh, and R. B. France. A systematic approach to generate inputs to test uml design models. In *Proceedings of 17th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Raleigh, North Carolina, USA, November 2006.

[23] Philip Samuel, Rajib Mall, and Pratyush Kanth. Automatic test case generation from uml communication diagrams. *Information and Software Technology*, 49(2):158–171, February 2007.

[24] Stephan Weileder and Bernd-Holger Schlingloff. Deriving input partitions from uml models for automatic test generation. In *Proceedings of Model-Driven Engineering, Verification and Validation*, pages 151–163. Springer-Verlag, 2008.

[25] Z. Dai. Model-driven testing with UML 2.0. In *Proceedings of the 2nd European Workshop on Model Driven Architecture*, 2004.

[26] Jean Hartmann, Marlon Viera, Herbert Foster, and Axel Ruder. A UML based approach to system testing. *Journal, Innovations in Systems and Software Engineering*, pages 12–24, 2005. Springer London.

[27] UML 2.0 Object Constraint Language OCL Specification. OMG document. Technical report, formal/06-05-01, 2006.

[28] Anneliese Amschler Andrews, Robert B. France, Sudipto Ghosh, and Gerald Craig. Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.

[29] A. Aho, R. Sethi, and J. Ullman. *Compilers - Principles, Techniques and Tools.* Addison Wesley, 1986.

[30] Zhenyi Jini A. Jefferson Offutt and Jie Pan. The dynamic domain reduction procedure for test data generation. *Software-Practice and Experience*, 29(2):167–193, 1999.

ABOUT THE AUTHORS

Ashalatha Nayak is an Assistant Professor in Department of Computer Science and Engineering at Manipal Institute of Technology, Manipal, India. She is pursuing her Ph.D. in the School of Information Technology, IIT Kharagpur in the area of Model Based Testing. Her research interests include program analysis and software testing. She obtained her B.Tech. and M.Tech. in Computer Science and Engineering from Mangalore University, Karnataka state, India. She can be reached at asha_nayak1@yahoo.com.

Debasis Samanta received his B. Tech. in Computer Science and Engineering from Calcutta University, M. Tech. in Computer Science and Engineering from Jadavpur University, Ph.D. in Computer Science and Engineering from Indian Institute of Technology, Kharagpur. He is currently an Assistant Professor in the School of Information Technology at the Indian Institute of Technology, Kharagpur. His research interests include biometric system, low power VLSI systems design, human computer interaction, and information system design. He can be reached at debasis.samanta.iitkgp@gmail.com.