A constraint-based framework for test case generation in method-level black-box unit testing

CHI-KUANG CHANG AND NAI-WEI LIN[*†]
Department of Computer Science and Information Engineering,
National Chung Cheng University,
Taiwan, R.O.C.

Automatic test case execution in test-driven development provides an excellent return on investment. However, test cases in test-driven development are usually designed manually. Manual acquisition of test cases is laborious, time-consuming, and error-prone. Model-based testing is a technique to automatically generate test cases from software models. Model-based test-driven development provides an opportunity to automate both test case generation and test case execution. This paper proposes and implements a constraint-based framework for automatic test case generation in method-level black-box unit testing. This framework uniformly solved the test case generation problem using constraint logic graphs and constraint logic programming. This framework effectively performs equivalence class partitioning and test coverage criteria management on constraint logic graphs, and simultaneously generates test input and expected output using constraint logic programming. This unifying constraint-based framework can serve as a nucleus for test case generation in model-based unit testing in the future, including method-level black-box, method-level white-box, and class-level unit testing.

Keywords: *constraint-based testing, black-box testing, unit testing, method-level unit testing, constraint satisfaction problem, constraint logic programming*

1. INTRODUCTION

Software unit testing is a process that includes the performance of test planning, the acquisition of a set of test cases, and the measurement of a test unit against its specifications [1]. Unit testing usually provides an excellent return on investment among various levels of testing [2]. However, the acquisition of a set of test cases in unit testing is a challenging task. In procedural programming, a unit can be an individual function or procedure in a module. In object-oriented programming, a unit is often an entire interface, such as a class, yet it can also be an individual method in the class. Method-level unit testing focuses on verifying state changes for a single method invocation. After method-level unit testing is completed, class-level unit testing can focus on verifying the state transitions for a sequence of method invocations. The separation of unit testing into method-level and class-level allows simpler management for unit testing.

In traditional software development, test cases in unit testing are usually designed when the unit is implemented or after the unit is implemented. In the test-driven development (TDD) approach [3], test cases in unit testing must be designed before the unit is implemented. TDD has several benefits and offers an excellent return on investment [4]. For example, developers acquire a deeper understanding of the unit requirements after designing the test cases. Developers can obtain faster feedback by promptly and repeatedly executing the test cases. Developers usually spend less time in debugging errors. Hence, the unit is of superior quality.

Automatic test case execution is crucial for TDD and is mature. However, test cases are usually designed manually in TDD. Manual acquisition of test cases is laborious, time-consuming, and error-prone. It is beneficial if test cases can be automatically generated instead. Model-based testing (MBT) provides a technique to automatically generate test cases from software models or specifications [5][6][7]. The architecture of model-based test case generation is shown in Figure 1. Test-platform-independent test data (or abstract test cases) are first generated automatically from the software model. Test-platform-specific test scripts (or concrete test cases) are then generated

[*] Correspondence to: Department of Computer Science and Information Engineering, National Chung Cheng University, 621 ChiaYi, Taiwan, R.O.C.
[†] E-mail: naiwei@cs.ccu.edu.tw

automatically from test-platform-independent test data and the test-platform templates. This paper will address the problem of automatic generation of test-platform-independent test data from software models. Therefore, by providing formal models, model-based TDD can automate both test case generation and test case execution.

Numerous modeling notations were proposed to specify software behaviors [5][6][7]. The most common approach applied in test case generation is using state-based specification languages [8], such as VDM [9], B[10], Z[11], JML[12], and UML/OCL [13][14]. We used the generic standardized modeling language, Unified Modeling Language (UML) and Object Constraint Language (OCL), as the modeling notation. UML can model various static structures and dynamic behaviors of a software system. In object-oriented programming, method specification is contained in the class specification. Therefore, we used the UML class diagram to describe the model of a class. A class diagram is used to define the attributes and methods of each class and the relationships among classes. OCL is primarily used to specify the behaviors of a method. In state-based specification languages, system states are denoted by a collection of state variables. The values of these state variables represent the current state of the system.

The behaviors of a method are characterized by the value changes of the state variables before and after method invocation. The behaviors of a method are specified by method preconditions and postconditions. A method precondition is the set of constraints that must be held before invoking a method to ensure the successful execution of the method invocation. A method postcondition is the set of constraints that must be held after the method invocation. The state change of the method invocation is usually specified in the method postconditions. In addition to method preconditions and postconditions, class invariants are used to specify the set of constraints that must be held by any object of the class during the life cycle of that object.

The set of constraints included in the model of a method specifies the behaviors regarding the relationships between the input and output of the method. Ideally, software testing ensures the correctness of all behaviors of the method. In practice, verifying each behavior of the method is usually impossible because of a large or infinite amount of behaviors. The first challenge in test case generation is partitioning the behaviors into a more manageable collection of equivalence classes to ensure that only a few representative test cases are generated for each equivalence class. However, the number of equivalence classes may still be large or infinite. The second challenge in test case generation is managing the coverage of equivalence classes of behaviors based on a test coverage criterion. A test case consists of a pair of test input and expected output. The third challenge in test case generation is generating test input for each representative test case, and the fourth challenge in test case generation is generating the corresponding expected output for each test input.

The test case generation in method-level black-box unit testing can be formulated as a collection of constraint satisfaction problems (CSPs). This paper proposes a constraint-based framework to uniformly overcome these four challenges, as shown in Figure 2. This framework overcomes these four challenges using a constraint representation, called constraint logic graph, and a constraint solving approach, called constraint logic programming (CLP) [15]. This framework uniformly overcomes the equivalence class partitioning and test coverage criteria management challenges based on constraint logic graph. This framework uniformly overcomes the test input generation and expected output generation challenges based on constraint logic programming.

The remainder of this paper is organized as follows: Section 2 demonstrates that the test case generation in method-level black-box unit testing can be formulated as a collection of CSPs. Section 3 introduces the proposed approach to enumerating CSPs according to UML/OCL specifications based on constraint logic graphs. Section 4 presents the mechanisms used for applying the CLP approach to determine solutions to the enumerated CSPs. Section 5 presents a prototype implementation of the framework. Section 6 provides a summary of related work. Finally, Section 7 provides our conclusion.

2. FORMULATING TEST CASE GENERATION AS CSPS

A method is a procedure that performs a specific task. The effects caused by executing the method depend on the state of the program and the arguments to the method. The effects may include a returned value or a change in the state of the program. The state of a program may include the instance variables of the receiver object, the static variables of classes, global variables, or input data.

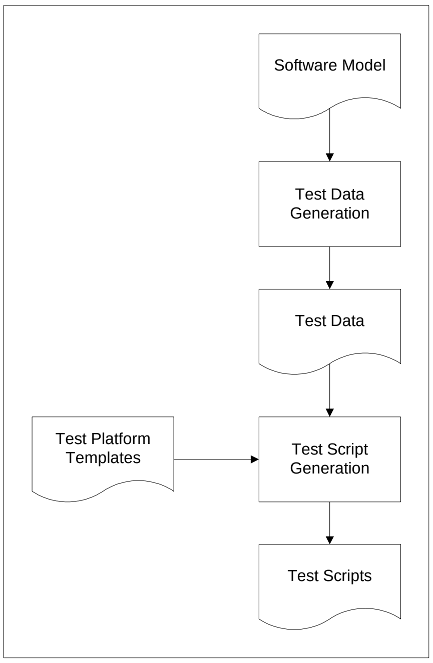


Figure 1. The architecture of model-based test case generation.

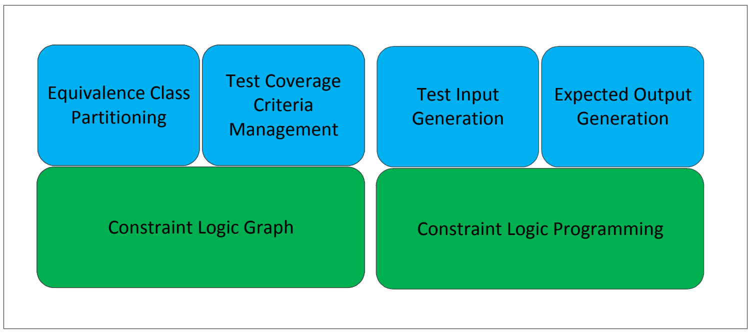

Equivalence Class Partitioning
Test Coverage Criteria Management
Test Input Generation
Expected Output Generation
Constraint Logic Graph
Constraint Logic Programming

Figure 2. The constraint-based framework for test case generation.

2.1 Method Specification

Let S denote the set of program states, A denote the set of arguments to the method, and R denote the set of returned values. A method m of a class C can be represented as a function

$$f_{C,m}: S \times A \rightarrow R \times S,$$

where $S \times A$ is the domain and $R \times S$ is the range of $f_{C,m}$.

A constructor c of C can be represented as a function

$$f_{C,c}: A \rightarrow S,$$

where A is the domain and S is the range of $f_{C,c}$. A constructor has no returned value and creates an object with an initial state. A constructor is a special case of a method. We assume each class has a constructor that can create objects at an appropriate state for testing.

The semantics of a method can be defined by three types of logical predicates or constraints: class invariants, method preconditions and method postconditions.

The class invariant for a class C specifies the constraints that must be held by any object of C during the life cycle of that object. The state of an object of C includes the instance variables and the static variables of C. The state of the program includes the state of an object. The state of an object of C does not include global variables, the static variables of classes other than C, nor input data. Let O be the set of states of an object of C. The class invariant for a class C can be represented as a relation

$$\Gamma_{C,inv}: O \rightarrow \{true, false\},$$

where $s \in O$ is in $\Gamma_{C,inv}$ or $\Gamma_{C,inv}(s) = True$, if s is a valid state of an object of C.

The method precondition for a method m of a class C specifies the constraints on the domain $S \times A$ of $f_{C,m}$ to successfully execute m. Therefore, the method precondition for a method m of a class C can be represented as a relation

$$\Gamma_{C,m,pre}: S \times A \rightarrow \{true, false\}$$

where $(s_{pre}, a) \in S \times A$ is in $\Gamma_{C,m,pre}$ if $f_{C,m}(s_{pre}, a)$ is defined, and $s_{pre} \in S$ is the state of the program at the moment that the execution of m begins.

The method postcondition for a method m of a class C specifies the constraints on the domain and range $S \times A \times R \times S$ of $f_{C,m}$ to successfully execute m. Therefore, the method postcondition for a method m of a class C can be represented as a relation

$$\Gamma_{C,m,post}: S \times A \times R \times S \rightarrow \{true, false\}$$

where $(s_{pre}, a, r, s_{post}) \in S \times A \times R \times S$ is in $\Gamma_{C,m,post}$ if $f_{C,m}(s_{pre}, a) = (r, s_{post})$, and $s_{post} \in S$ is the state of the program when the execution of m is completed.

2.2 Test Case Specification

Test case generation for a method can be formulated as a collection of CSPs as follows: a CSP involves a list of variables that are defined on finite domains of values and a set of constraints that restrict the values that the variables can simultaneously use [16]. A solution to a CSP is a set of values assigned to variables that satisfy all of the constraints. A CSP can be represented as a triple $< V, D, \Gamma >$, where V denotes the finite set of CSP variables, D is the set of domains (one for each variable), and Γ is the set of constraints over the variables.

A test case for a method consists of a pair of test input and expected output. Each $(s_{pre}, a) \in S \times A$ is a candidate for test input, and each $(r, s_{post}) \in R \times S$ is a candidate for expected output. A suite of test cases for a method consists of test cases for valid test input and test cases for invalid test input.

Test cases for valid test input ensure that the method behaves correctly when it incurs valid input data. The problem of identifying a test case for valid test input can be formulated as the following CSP T_v:

$$< (V_{s,pre} \cup V_a \cup V_r \cup V_{s,post}),$$

$$(D_{s,pre} \times D_a \times D_r \times D_{s,post}),$$

$$(\Gamma_{C,inv,pre} \wedge \Gamma_{C,m,pre} \wedge \Gamma_{C,m,post} \wedge \Gamma_{C,inv,post}) >,$$

where, $V_{s,pre}$, with the domain $D_{s,pre}$, is the set of variables denoting the state of the program when the execution of a method m begins. V_a, with the domain D_a, is the set of variables denoting the arguments to m. V_r, with the domain D_r, is the variable denoting the value returned from m. $V_{s,post}$, with the domain $D_{s,post}$, is the set of variables denoting the state of the program when the execution of m is completed. $\Gamma_{C,inv,pre}$ is the class invariant with respect to the variables in $V_{s,pre}$. $\Gamma_{C,inv,post}$ is the class invariant with respect to the variables in $V_{s,post}$. A solution to T_v is a test case candidate for valid test input.

Test cases for invalid test input ensure that the method can behave gracefully when it incurs invalid test input. In general, the method throws an exception when it incurs invalid test input. The problem of identifying a test case for invalid test input can be represented as the following CSP T_i :

$$< (V_{s,pre} \cup V_a), \left(D_{s,pre} \times D_a\right), \left(\Gamma_{C,inv,pre} \wedge \sim\Gamma_{C,m,pre}\right) >.$$

A solution to T_i is a test case candidate for invalid test input. Identifying test cases for invalid test input is similar to identifying test cases for valid input data. We only discuss identifying test cases for valid test input.

3. ENUMERATING CSPS BASED ON CONSTRAINT LOGIC GRAPHS

The domain D of the CSP, $< V, D, \Gamma >$, used for identifying a test case for a method m of a class C may be large. Test case generation can generate a small suite of test cases that effectively ensure the quality of the program. Equivalence class partitioning is the most widely used technique for generating a small and effective suite of test cases.

In the equivalence class partitioning technique, the domain $D_{s,pre} \times D_a$ of $f_{C,m}$ of a method m of a class C is partitioned into a number of equivalence classes. These equivalence classes are mutually exclusive and collectively exhaustive. The elements belonging to the same equivalence class induce the method to behave “equivalently” regarding test-effectiveness; in other words, in principle, one element (or a small number of elements) is chosen as the representative of an equivalence class and is included in the test suite.

Consequently, by using the equivalence class partitioning technique, the CSP for the test case generation can be partitioned into a collection of sub-CSPs as follows: let

$$\Gamma = \Gamma_1 \vee \ldots \vee \Gamma_n$$

be the constraint of the CSP represented as a disjunction of a set of constraints. The CSP for the test case generation can be partitioned into n sub-CSPs, $< V, D, \Gamma_i >, 1 \leq i \leq n$. The set of elements in D that satisfies Γ_i forms an equivalence class.

For example, the CSP for the test case generation of a method m of a class C can be initially partitioned into two sub-CSPs, T_v and T_i using

$$\Gamma = \left(\Gamma_{C,inv,pre} \wedge \Gamma_{C,m,pre} \wedge \Gamma_{C,m,post} \wedge \Gamma_{C,inv,post}\right) \vee \left(\Gamma_{C,inv,pre} \wedge \sim\Gamma_{C,m,pre}\right).$$

These sub-CSPs can then be further partitioned.

The number of sub-CSPs partitioned for a method may be large or infinite. Therefore, numerous test coverage criteria are proposed to limit the partitioning of equivalence classes. This paper proposes the use of constraint logic graphs to facilitate the application of equivalence class partitioning and test coverage criteria management.

3.1 Constraint Logic Graph

A constraint logic graph is a succinct graphical representation of a clause of (possibly infinite) constraints. A constraint logic graph is represented as a directed graph with seven types of nodes: the initial node, the final node, constraint nodes, disjunction-initial nodes, disjunction-final nodes, conjunction nodes, and iterated conjunction nodes. A constraint logic graph possesses a unique initial node, depicted as a filled circle, and a unique final node, depicted as a filled double circle. The initial node has only outgoing edges. The final node has only incoming edges. A constraint is represented by a constraint node, depicted as a rectangle. The constraint in a constraint node may be a simple constraint, a constraint abstraction, or a negation of a simple constraint or a constraint abstraction. A simple constraint is a constraint without conjunction or disjunction operators and a constraint abstraction is the constraint for a method.

The disjunction of two constraints is represented by a pair of disjunction-initial and disjunction-final nodes connecting the two corresponding constraint nodes. A disjunction-initial node is depicted as a diamond and a disjunction-final node is depicted as a circle. The conjunction of two constraints is represented by a conjunction node connecting the two corresponding constraint nodes. A conjunction node is depicted as a pentagon. An iterated-conjunction node is used to succinctly represent a conjunction of a series of constraints with indexed variable names. An iterated conjunction node is depicted as a hexagon.

An OCL expression can be represented by a constraint logic graph, as follows: a simple OCL constraint is represented by a constraint node. *IfExp*, *IterateExp* and *OperationCallExp* are three major OCL constructs that form compound constraints. An *IfExp* expression has the general form of

$$\text{if } \Gamma_{cond} \text{ then } \Gamma_{true} \text{ else } \Gamma_{false} \text{ endif,} \tag{1}$$

which denotes the following constraint

$$(\Gamma_{cond} \land \Gamma_{true}) \lor (\neg\Gamma_{cond} \land \Gamma_{false}). \tag{2}$$

Thus, an *IfExp* expression can be represented in the constraint logic graph, as illustrated in Figure 3, where the two conjunctive paths are connected by a diamond disjunction-initial node and a circular disjunction-final node to form a disjunction.

An *IterateExp* expression has the general form of

collection->iterate(*e* : *type1*; *r : type2 = initial_expr* | *body_expr*) (3)

where e is the element variable and r is the accumulator variable. The accumulator r is initialized to *initial_expr*. The *IterateExp* expression applies *body_expr* to each element e of the collection *collection*. Such an *IterateExp* expression denotes the following constraint

$$\Gamma_{initial} \land \left(\bigwedge_{i=1}^{n} \Gamma_{body}(i)\right) \land \Gamma_{final} \tag{4}$$

where

$$\Gamma_{initial} \equiv (n = collection \to size()) \land (r_0 = initial\_expr) \tag{5}$$

$$\Gamma_{body}(i) \equiv (i \le n) \land (e_i = collection \to at(i)) \land (r_i = body\_expr\,[r/r_{i-1}, e/e_i]) \tag{6}$$

$$\Gamma_{final} \equiv (i > n) \land (r = r_n) \tag{7}$$

Differing versions of e_i and r_i are used for variables e and r in various iterations of i. The expression $body\_expr\,[r/r_{i-1}, e/e_i]$ indicates that the variables r and e in $body\_expr$ are substituted by the variables r_{i-1} and e_i, respectively.

An *IterateExp* expression can be represented in the constraint logic graph, as illustrated in Figure 4, where the iterated conjunctive paths are connected by a hexagonal iterated-conjunction node to form a conjunction.

An *OperationCallExp* for a user-defined method represents the constraint abstraction of a method invocation, which is depicted as a normal constraint node in the constraint logic graph. The constraint abstraction is composed of the complete set of OCL constraints of the method; that is, the compound constraint of the class invariant before the invocation occurs, the precondition and the postcondition of the method, and the class invariant after the invocation is completed.

Each complete path displayed in the constraint logic graph corresponds to a sub-CSP (or an equivalence class) of the CSP. If iterated-conjunction nodes are present in the graph, the number of paths may be infinite. To address this problem, paths can be enumerated in a breadth-first-traversal manner until the test coverage criterion is met. Breadth-first-traversal can prevent infinite looping and systematically enumerates from short to long paths. However, an enumerated path may correspond to an unsolvable CSP. In this situation, the enumerated path is discarded, and the path enumeration process continues. An unsolvable CSP is detected when the constraint solver (described in the next section) fails to determine a solution, or the resolution time exceeds a predetermined threshold.

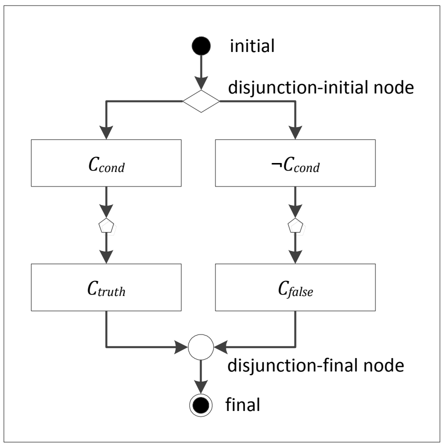


Figure 3. The constraint logic graph for an *IfExp* expression.

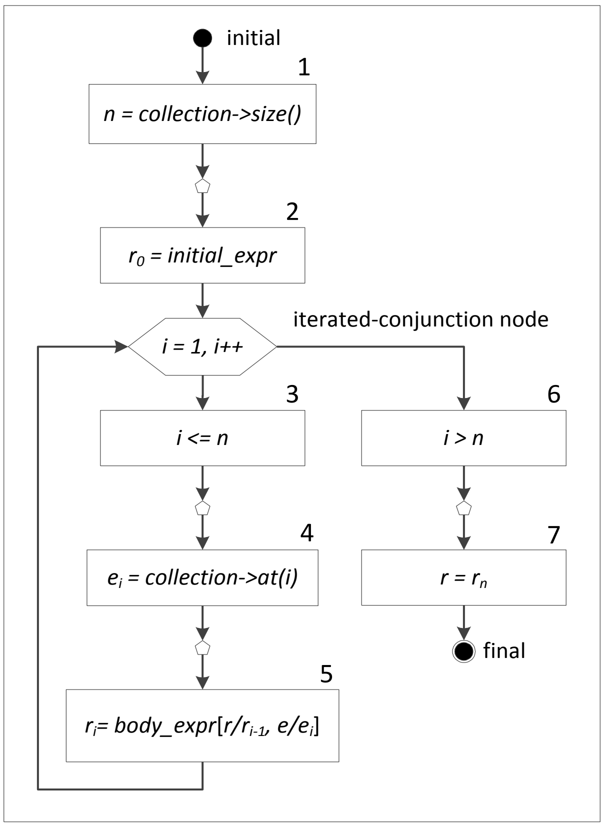


Figure 4. The constraint logic graph for an IterateExp expression.

3.2 A Running Example

The IntRange class is used as an example to demonstrate the generation of constraint logic graphs and the generation of sub-CSPs. The IntRange class models a range of integers. Figure 5 displays the UML class diagram and

Figure **6** lists the OCL definitions associated with IntRange. The method contains() is used as an example. The problem of identifying a test case for the valid test input of method contains() can be formulated as the following CSP:

$$< \{start_{pre}, end_{pre}\} \cup \{v\} \cup \{result\} \cup \{start_{post}, end_{post}\},$$
$$(Integer, Integer) \times (Integer) \times (Boolean) \times (Integer, Integer),$$
$$\Gamma_{IntRange,inv,pre} \wedge \Gamma_{IntRange,contains,pre} \wedge \Gamma_{IntRange,contains,post} \wedge \Gamma_{IntRange,inv,post} >$$

The constraint logic graphs of invariants $\Gamma_{IntRange,inv,pre}$ and $\Gamma_{IntRange,inv,post}$ contain only a constraint node of a simple constraint. The method contains() does not have a precondition. Figure 7 displays the constraint logic graph of the postcondition $\Gamma_{IntRange,contains,post}$, which is a conjunction of two postcondition constraints.

The following path enumeration process demonstrates the test case generation for all-branch test coverage criterion. The first enumerated path is {1, 2, 3, 10, 11, 12, 13, 14}, which depicts a path traversing zero iteration over the *IterateExp* expression. The constraint of the sub-CSP corresponding to this path is derived as follows:

$$\begin{gathered}(start_{pre} <= end_{pre}) \wedge \\ (L_0 = \{start_{pre}, start_{pre} + 1, \ldots, end_{pre}\}) \wedge \\ (n = L_0\text{->}size()) \wedge (answer_0 = 0) \wedge \\ (1 > n) \wedge (answer = answer_0) \wedge (result = answer) \wedge \\ (start_{post} = start_{pre}) \wedge (end_{post} = end_{pre}) \wedge \\ (start_{post} <= end_{post}).\end{gathered} \tag{8}$$

Because an IntRange object contains at least one element, the inequality constraint of (1 > n) is constantly false. Thus, the CSP corresponding to this path is an unsolvable CSP.

The second enumerated path is {1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14}, which depicts a path traversing one iteration over the *IterateExp* expression and traversing the true branch of the *IfExp* expression. The constraint corresponding to this path is presented as follows:

$$\begin{gathered}(start_{pre} <= end_{pre}) \wedge \\ (L_0 = \{ start_{pre}, start_{pre} + 1, \ldots, end_{pre}\}) \wedge \\ (n = L_0\text{->}size()) \wedge (answer_0 = \text{false}) \wedge \\ (1 <= n) \wedge (e_1 = L_0\text{->}at(1)) \wedge (e_1 = v) \wedge (answer_1 = 1) \wedge \\ (2 > n) \wedge (answer = answer_1) \wedge (result = answer) \wedge \\ (start_{post} = start_{pre}) \wedge (end_{post} = end_{pre}) \wedge \\ (start_{post} <= end_{post}).\end{gathered} \tag{9}$$

One solution to Constraint (9) is

$$\{start_{pre} = 1, end_{pre} = 1, v = 1, result = \text{true}, start_{post} = 1, end_{post} = 1\}.$$

The third enumerated path is {1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 14}, which depicts a path traversing one iteration over the *IterateExp* expression and traversing the false branch of the *IfExp* expression. One solution to the third path is

$$\{ start_{pre} = 1, end_{pre} = 1, v = 2, result = \text{false}, start_{post} = 1, end_{post} = 1\}.$$

Because all of the branches have been traversed, the path enumeration process terminates.

Because a precondition is not specified in the method of contains(), we used the constructor IntRange() as an example to generate test cases for invalid test input. The problem of identifying a test case for invalid test input of the constructor IntRange() can be formulated as the following CSP:

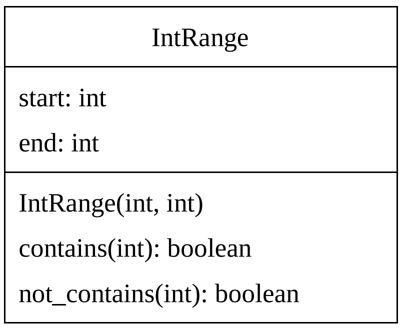


Figure 5. The class diagram for class IntRange.

```
package intRange

context IntRange
inv: start <= end

context IntRange::IntRange(first:Integer, last:Integer):
pre: first <= last
post: start = first and end = last

context IntRange::contains(v: Integer): Boolean
post return_value:
     result = Sequence{start..end}->iterate
     (e: Integer; answer: Boolean = false |
         if e=v then true else answer endif
     )
post state_update:
     start = start@pre and end = end@pre

context IntRange::not_contains(v: Integer): Boolean
post return_value:
      if self.contains(v) then
            result = false
      else
            result = true
      endif
post state_update:
     start = start@pre and end = end@pre
```

Figure 6. The OCL specification for class IntRange.

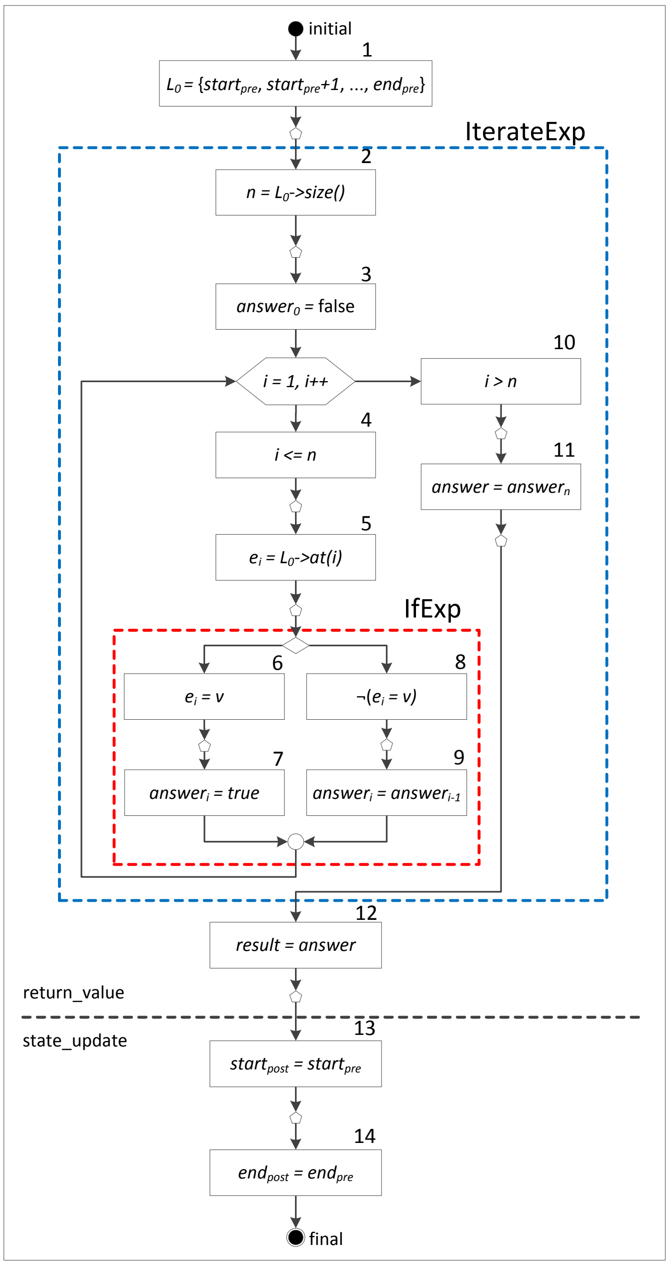


Figure 7. The constraint logic graph for the postcondition of the method contains().

$$< [start_{pre}, end_{pre}] \cup [first, last],$$
$$(Integer, Integer) \times (Integer, Integer),$$
$$\Gamma_{intRange,inv,pre} \wedge \neg\Gamma_{intRange,intRange,pre} >.$$

The generation of compound constraints from the negation constraint of $\neg\Gamma_{intRange,intRange,pre}$ is achieved by negating the compound constraints generated from $\Gamma_{intRange,intRange,pre}$. The constraint logic graph of $\Gamma_{intRange,intRan,pre}$ displays two simple constraint nodes connected with a conjunction node. Only one path is generated; thus, the compound constraint for test cases of invalid test input is:

$$(start_{pre} <= end_{pre}) \wedge (\neg\ (first <= last)). \qquad (10)$$

A possible solution to this CSP is { $start_{pre}$ = 1, end_{pre} = 1, *first* = 1, *last* = 0}

The method not_contains() contains an invocation to the method contains() which is represented in the constraint logic graph as a constraint node containing a constraint abstraction. Two paths are enumerated from the constraint logic graph and the associated compound constraints are Constraint (11):

$$(start_{pre} <= end_{pre}) \wedge contains(v) \wedge (result = \text{false}) \wedge (start_{post} = start_{pre}) \wedge (end_{post} = end_{pre}) \wedge (start_{post} <= end_{post}) \qquad (11)$$

and Constraint (12):

$$(start_{pre} <= end_{pre}) \wedge (\neg contains(v)) \wedge (result = \text{true}) \wedge (start_{post} = start_{pre}) \wedge (end_{post} = end_{pre}) \wedge (start_{post} <= end_{post}) \qquad (12)$$

The possible solutions to Constraints (11) and (12) are {$start_{pre}$= 1, end_{pre} = 1, *v* = 1, *result* = false, $start_{post}$ = 1, end_{post} = 1} and {$start_{pre}$ = 1, end_{pre} = 1, v = 0, *result* = true, $start_{post}$ = 1, end_{post} = 1}, respectively.

4. SOLVING CSPS BASED ON CONSTRAINT LOGIC PROGRAMMING

Constraint logic programming (CLP) is a widely used approach for solving CSPs [15]. CLP is a combination of two declarative programming paradigms: logic programming and constraint programming. The users declaratively describe the problem variables, the domains of the variables, and the constraints over the variables. The CLP system can then determine the values of the variables that satisfy the constraints.

We converted the constraints expressed in OCL into the CLP predicates in the ECLiPSe system. The ECLiPSe system is based on Prolog and provides solvers for several types of constraints, including arithmetic constraints over finite domains, finite set constraints, linear rational constraints, and interval reasoning over non-linear constraints [17]. For each native OCL operation, a corresponding CLP predicate is implemented into the OCL2CLP library.

The converted CLP predicates are composed of three portions: the declaration of variable domains, the set of constraints, and the invoking of constraint solvers. The set of constraints in the CLP predicates are generated from the constraint nodes along the enumerated complete paths displayed in the constraint logic graph. Simple OCL constraints are directly mapped into the corresponding CLP constraint expressions. Take the CSP of Constraint (9) as an example. The generated CLP predicates are displayed in Figure 8. The domain of an integer-type variable is assumed to be from -32768 to 32767. The domain of a Boolean-type variable contains the values 0 and 1. The variable names start with a capital letter in CLP predicates. Therefore, the listed CSP variables must be renamed accordingly. A query test_intRange_contains([Start_pre, End_pre], [Arg], [Result], [Start, End]) to predicate test_intRange_contains/4 returns the solution{Start_pre = -32768, End_pre = -32768, Arg = -32768, Result = 1, Start = -32768, End = -32768}.

Predicate labeling/1 instantiates all variables to the values of the respective domains. Between the generated CLP predicates, predicate ocl2clp_new_integer_sequence/3 is a function in the OCL2CLP library that returns an integer sequence based on the starting integer and the final integer of an integer range. Predicate ocl2clp_sequence_length/3 is a function in the OCL2CLP library that returns the length of a sequence. Predicate ocl2clp_sequence_nth1/3 is a function in the OCL2CLP library that returns the *n*th element of a sequence.

```
% include
:- lib(ic).
:- lib(listut).

test_intRange_contains([Start_pre, End_pre], [Arg], [Result], [Start, End]) :-
% Declaration of variable domains
   [Start_pre, End_pre, Arg, Start, End] :: -32768 .. 32767,
   Result :: 0 .. 1,
% Set of constraints
   Start_pre #=< End_pre,
   ocl2clp_new_integer_sequence(Start_pre, End_pre, L0),
   ocl2clp_sequence_length(Seq, N),
   Answer_0 #= 0,
   1 #=< N,
   ocl2clp_sequence_nth1(1, Seq, E0),
   E0 #= Arg,
   Answer_1 #= 1,
   2 #> N,
   Answer #= Answer_1,
   Result #= Answer,
   Start #= Start_pre,
   End #= End_pre,
   Start #=< End,
% Invoking of the constraint solvers
   labeling([Start_pre, End_pre, Arg, Result, Start, End]).

ocl2clp_new_integer_sequence(S, E, L):-
   numlist(S, E, L).

ocl2clp_sequence_length(Seq, N):-
   length(Seq, N).

ocl2clp_sequence_nth1(N, Seq, E):-
   nth1(N, Seq, E).
```

Figure 8. The CLP predicate corresponding to Constraint (9) for method contains().

```
%include
:- lib(ic).
:- lib(listut).

test_intRange_not_contains([Start_pre, End_pre], [Arg], [Result], [Start, End]):-
   [Start_pre, End_pre, Arg, Start, End] :: -32768 .. 32767,
   Result :: 0 .. 1,
   Start_pre #=< End_pre,
   intRange_contains([Start_pre, End_pre], [Arg], [Result0], [Start, End]),
   Result0 #= 1,
   Result #= 0,
   Start #= Start_pre,
   End #= End_pre,
   labeling([Start_pre, End_pre, Arg, Result, Start, End]).

intRange_contains([Start_pre, End_pre], [Arg], [Result], [Start, End]) :-
   ocl2clp_new_integer_sequence(Start_pre, End_pre, Seq),
   eval_iterate_0(Answer, Seq, [], [Start_pre, End_pre], [Arg], [Result], [Start, End]),
   Result #= Answer,
   Start #= Start_pre,
   End #= End_pre.

eval_iterate_0(Val, Col, [], [Start_pre, End_pre], [Arg], [Result], [Start, End]):-
   eval_iterate_0(Col, 0, Val, [], [Start_pre, End_pre], [Arg], [Result], [Start, End]).

eval_iterate_0([], Val, Val, [], [Start_pre, End_pre], [Arg], [Result], [Start, End]).
eval_iterate_0([E|Es], Val0, Val, [], [Start_pre, End_pre], [Arg], [Result], [Start, End]) :-
   eval_if_0(Val1, [E, Val0], [Start_pre, End_pre], [Arg], [Result], [Start, End]),
   eval_iterate_0(Es, Val1, Val, [], [Start_pre, End_pre], [Arg], [Result], [Start, End]).

eval_if_0(Val, [Ele, Acc], [Start_pre, End_pre], [Arg], [Result], [Start, End]):-
   Arg #= Ele, Val #= 1.
eval_if_0(Val, [Ele, Acc], [Start_pre, End_pre], [Arg], [Result], [Start, End]):-
   Arg #\= Ele, Val #= Acc.

ocl2clp_new_integer_sequence(S, E, L):-
   numlist(S, E, L).
```

Figure 9. The CLP predicate corresponding to Constraint (11) for method not_contains().

Method not_contains() has a method invocation to contains(), which must correspond to a constraint abstraction. A constraint abstraction corresponds to the set of OCL constraints on the entire constraint logic graph instead of the set of OCL constraints on a single path. The CLP predicate for a constraint abstraction can be generated top-down along the respective OCL parse tree. Take the CSP of Constraint (11) as an example. The generated CLP predicates are displayed in Figure 9. A query test_intRange_not_contains([Start_pre, End_pre], [Arg], [Result], [Start, End]) to predicate test_intRange_not_contains/4 returns the solution {Start_pre = -32768, End_pre = -32768, Arg = -32768, Result = 0, Start = -32768, End = -32768}.

The CLP predicate C_m/4 for a method m of a class C has the following form:

C_m($V_{s,pre}$, V_a, V_r, $V_{s,post}$),

where $V_{s,pre}$, V_a, V_r, and $V_{s,post}$ are the variables of the corresponding CSP. A simple constraint is then converted. An *IfExp* expression is converted into a predicate eval_if_n/6 as follows:

eval_if_n(Val, Local, $V_{s,pre}$, V_a, V_r, $V_{s,post}$),

where Val is the Boolean value of the *IfExp* expression, and Local is the set of variables in the local scope of the method other than V_a and V_r. The predicate eval_if_n/6 contains a clause corresponding to the true branch and a clause corresponding to the false branch. A predicate eval_if_n/6 is generated dynamically for each *IfExp* expression. Therefore, the *n* in predicate eval_if_n/6 is an integer used to assemble a unique predicate name for each *IfExp* expression.

An *IterateExp* expression is converted into a predicate eval_ iterate_n/7 as follows:

eval_ iterate_*n*(Val, Col, Local, $V_{s,pre}$, V_a, V_r, $V_{s,post}$),

where Val is the accumulator variable and Col is the collection of the *IterateExp* expression, and Local is the set of variables in the local scope of the method other than V_a and V_r. A predicate eval_ iterate_n/7 is also generated dynamically for each *IterateExp* expression. To efficiently compute the accumulator variable, the predicate eval_ iterate_n/7 calls the helper predicate eval_ iterate_n/8:

eval_ iterate_*n*(Col, Val0, Val, Local, $V_{s,pre}$, V_a, V_r, $V_{s,post}$),

where Val0 and Val are the accumulating value and the final value of the accumulator variable, respectively. The local variables E and Val0 of the predicate eval_iterate_0/8 are included in the argument Local to the predicate eval_if_0, as displayed in Figure 9.

5. A PROTOTYPE IMPLEMENTATION

We implemented a prototype system that can automatically generate method-level unit testing test cases for both Java and C++ programming languages. The prototype system uses the Eclipse Modeling Framework (EMF) to specify the UML model and OCL constraints. Ecore Tools SDK was applied to parse the OCL constraints to construct constraint logic graphs. The supported testing platforms were JUnit [18] for Java and CUTE [19] for C++. This system can be integrated into Eclipse [20] to automate the test-driven development process. Figure 10 displays the architecture of the prototype system.

This prototype system accepts UML and OCL documents as inputs and generates test scripts for specific test-platforms as the output. This system contains five major components: the test coverage criteria manager, the path enumerator, the CLP generator, the CLP executor and the test script generator. The test coverage criteria manager is the kernel component, which accepts software models and converts them into constraint logic graphs. The test coverage criteria manager also monitors the coverage of test cases and determines when the path numeration process terminates. The path enumerator systematically enumerates complete paths on the constraint logic graphs. The CLP generator subsequently converts each complete path into a CLP predicate. The converted CLP predicate is submitted to the CLP executor to compute test input and expected output. Finally, the test script generator generates testing-platform-specific test scripts for various testing platforms.

This prototype system has a simple GUI. Figure 11 shows an example screenshot of this prototype system. This version produces test scripts for JUnit testing framework. The "Feasible Paths" window shows the enumerated feasible complete paths on the constraint logic graph. The "CLP Programs" window shows the converted CLP predicates for the corresponding feasible complete paths. The "Test Data" window shows the computed test data for the corresponding CLP predicates. The "JUnit Test Class" window shows the generated Java test class for the Java class under test.

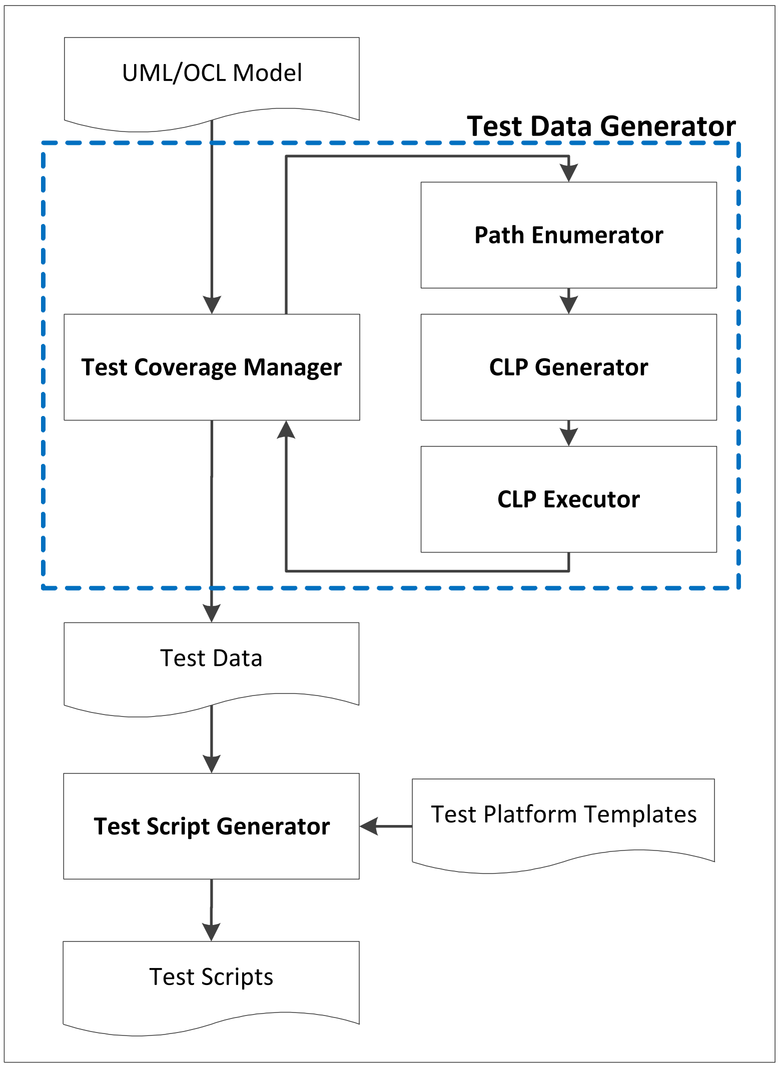
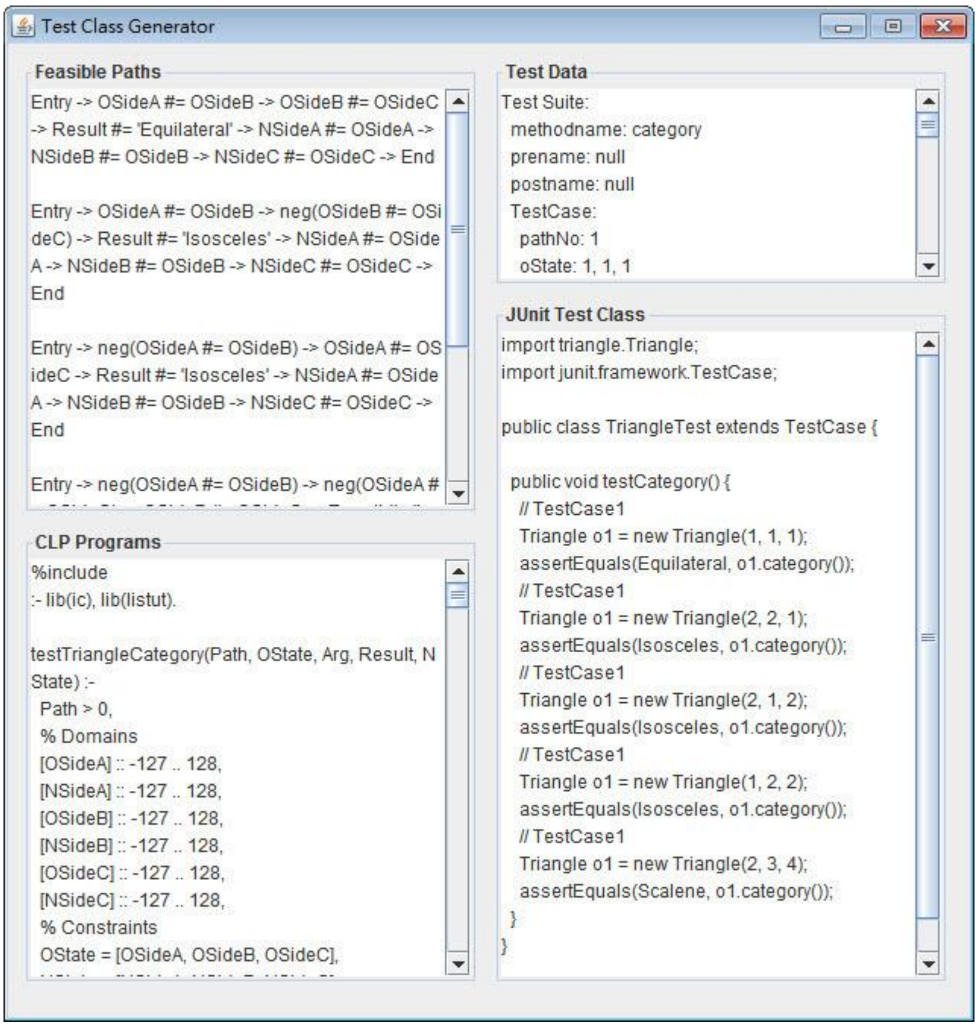


Figure 10. The architecture of the prototype system.



```
Entry -> OSideA #= OSideB -> OSideB #= OSideC
-> Result #= 'Equilateral' -> NSideA #= OSideA ->
NSideB #= OSideB -> NSideC #= OSideC -> End

Entry -> OSideA #= OSideB -> neg(OSideB #= OSi
deC) -> Result #= 'Isosceles' -> NSideA #= OSide
A -> NSideB #= OSideB -> NSideC #= OSideC ->
End

Entry -> neg(OSideA #= OSideB) -> OSideA #= OS
ideC -> Result #= 'Isosceles' -> NSideA #= OSide
A -> NSideB #= OSideB -> NSideC #= OSideC ->
End

Entry -> neg(OSideA #= OSideB) -> neg(OSideA #
```

```
Test Suite:
 methodname: category
 prename: null
 postname: null
 TestCase:
  pathNo: 1
  oState: 1, 1, 1
```

```
%include
:- lib(ic), lib(listut).

testTriangleCategory(Path, OState, Arg, Result, N
State) :-
  Path > 0,
  % Domains
  [OSideA] :: -127 .. 128,
  [NSideA] :: -127 .. 128,
  [OSideB] :: -127 .. 128,
  [NSideB] :: -127 .. 128,
  [OSideC] :: -127 .. 128,
  [NSideC] :: -127 .. 128,
  % Constraints
  OState = [OSideA, OSideB, OSideC],
```

```
import triangle.Triangle;
import junit.framework.TestCase;

public class TriangleTest extends TestCase {

 public void testCategory() {
  // TestCase1
  Triangle o1 = new Triangle(1, 1, 1);
  assertEquals(Equilateral, o1.category());
  // TestCase1
  Triangle o1 = new Triangle(2, 2, 1);
  assertEquals(Isosceles, o1.category());
  // TestCase1
  Triangle o1 = new Triangle(2, 1, 2);
  assertEquals(Isosceles, o1.category());
  // TestCase1
  Triangle o1 = new Triangle(1, 2, 2);
  assertEquals(Isosceles, o1.category());
  // TestCase1
  Triangle o1 = new Triangle(2, 3, 4);
  assertEquals(Scalene, o1.category());
 }
}
```

Figure 11, An execution result

6. RELATED WORK

This section reviews related work that applies the constraint-based approach to testing.

6.1 Fault-Based Testing

DeMillo and Offutt are the pioneers to use the constraint-based approach to testing [21]. They applied the constraint-based approach to automatically generate test inputs in fault-based testing. Fault-based testing generates test inputs to show the presence or absence of most common faults that a programmer might introduce [22][23]. Fault-based testing generates a set of mutant programs each of which contains a common fault represented as a simple syntactic change to the program under test. They derived algebraic constraints from each mutant program to specify the conditions under which the execution of this program can reach the faulty point. These algebraic constraints are reduced into disjunctive normal form. Each conjunctive clause corresponds to a unique path. They implemented a constraint solver to solve these algebraic constraints. The solution to a conjunctive clause is the test input that drives the program to the faulty point through the corresponding path. The expected output is usually generated manually.

Aichernig and Pari Salas applied the constraint-based approach to automatically generate test inputs from model-based specifications in fault-based testing [24]. They generated mutant specifications instead of mutant programs. They generated mutant OCL preconditions and postconditions. They derived algebraic constraints from each mutant specification to specify the conditions under which the execution of this program can discover the corresponding fault.

6.2 White-Box Testing

There are a number of researches that applied the constraint-based approach to automatically generate test inputs in white-box testing. Gotlieb, Botella, and Rueher [25] proposed a method that first transforms the C program under test into static single assignment form so that each variable in the program is assigned statically at most once. This method then derives a system of constraints for different execution paths in the control flow graph using symbolic execution. Finally, this method uses dedicated constraint solvers to automatically generate the test input for each feasible execution path. The expected outputs for white-box testing are usually generated manually. They later developed techniques to handle more complex data types such as floating point numbers [26] and structural data types [27].

Meudec [28] used symbolic execution to collect a system of constraints for different execution paths in Ada programs. He then used constraint logic programming to automatically generate the test input for each feasible execution path. Lembeck et al. [29] used an approach similar to Meudec for Java byte codes. Their system implemented a symbolic Java virtual machine and can support def-use chain coverage.

Williams et al. [30] used code instrumentation to collect the path predicate on a path for the concrete execution of a test input. Initially, the test input is selected arbitrarily. They then used constraint logic programming to compute the set of input values covered by this path predicate. They then selected the next test input from the domain outside of this set of input values. This process continues until the domain becomes empty. Sen, Marinov, and Agha [31] used an approach similar to Williams et al. and also handled dynamic data structures using pointers. They used a logic input map to represent inputs as a collection of scalar symbolic variables and construct constraints on these variables by symbolic execution.

Milicevic et al. [32] proposed a tool for generating structurally complex test inputs for Java. The user needs to define a Java method to specify the bounds and invariants of the data structure. Senni and Fioravanti [33] proposed another approach for generating structurally complex test inputs. They used constraint logic programming to specify the bounds and invariants of the data structure.

6.3 Black-Box Testing

There are a number of researches that applied the constraint-based approach to automatically generate test cases in black-box testing. Dick and Faivre presented a technique for automatic test case generation from the VDM specification [34]. They symbolically transformed the VDM specification for the program under test into a first-order predicate calculus in disjunctive normal form. Each conjunctive clause in the disjunctive normal form corresponds to an equivalence class. They also considered the generation of a sequence of method calls to bring the program into an appropriate system state for testing.

The AutoFocus testing tool is a test case generation framework based on the model simulation using constraint logic programming [35][36]. AutoFocus supports the modeling and analyzing of the structure and behavior of distributed, reactive, and timed computer-based systems. AutoFocus uses UML-like structural diagrams and state diagrams to describe the interactions between components. First, AutoFocus translates these diagrams into CLP predicates, and subsequently executes CLP predicates to generate test sequences. AutoFocus mainly generates test cases for class-level unit testing.

The BZ testing tool is a test case generation framework based on the symbolic animation using constraint logic programming [37][38]. By using the BZ testing tool, a software system can be specified in B, Z, OCL, or JML modeling languages. A transition from one state to another state is formulated as a CSP. The model is first translated into a Prolog-like intermediate format: BZPE. This intermediate format is then animated using constraint solvers CLP(FD) [39] and CLPS-BZ [40].

7. FUTURE WORK AND CONCLUSION

This paper proposes and implemented a constraint-based framework to generate test cases for method-level black-box unit testing. This framework uniformly solved the test case generation problem using constraint logic graphs and constraint logic programming. This framework effectively performs equivalence class partitioning and test coverage criteria management on the constraint logic graphs, and simultaneously generates both test input and expected output using constraint logic programming. This constraint-based framework can serve as a nucleus for test case generation in model-based unit testing. This framework was used to generate test cases for black-box method-level unit testing. In the future, this framework will be extended to generate test cases for method-level white-box unit testing and class-level unit testing.

The extension of this framework to method-level white-box unit testing is outlined as follows. The method under test can first be transformed into a method in static single assignment form. The control flow graph of the method in static single assignment form can then be converted into a constraint logic graph by replacing each assignment with an equality constraint. With a constraint logic graph representing the implementation of the method, the framework can automatically generate test cases for method-level white-box unit testing. This has been addressed in a number of related studies [25][28][29].

The extension of this framework to class-level unit testing is outlined as follows. UML state diagrams and OCL can be used as the specification language. The UML state diagram for a class can be converted into a constraint logic graph by transforming each transition in the state diagram into a constraint node with a constraint abstraction that represents a method invocation. Instead of generating a pair of test input and expected output for a single method invocation in method-level testing, a sequence of pairs can be generated for the sequence of method invocations on a complete path in constraint logic graph. This has been addressed in a number of related studies [34][35][37].

ACKNOWLEDGMENTS

This work was supported in part by the National Science Council of R.O.C. under grant number NSC-97-2221-E-194-036, NSC-98-2221-E-194-042, NSC-99-2628-E-194-006, NSC-100-2221-E-194-033. We are grateful to our colleagues: Yi-Tin Hu, Chih-Yuan Lin, Ming-Bin Ho and Yueh-Hsun Lee for their helps in the implementation of the prototype system.

REFERENCES

1. IEEE Standards Board, “IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987,” *IEEE Standards: Software Engineering*, Vol. Two: Process Standards, 1999.
2. D. Graham, M. Fewster, *Experiences of Test Automation: Case Study of Software Test Automation*, Addison Wesley, 2012.
3. B. Bezier, *Software Testing Techniques*, Second Edition, Dreamtech, 2003.
4. B. George, L. Williams, “An initial investigation of test driven development in industry,” in *Proceedings of the 2003 ACM symposium on Applied computing*, 2003, pp. 1135-1139.
5. C.A.D. Neto, R. Subramanyan, M. Vieira, G.H. Travassos, “A survey on model-based testing approaches: A systematic review,” in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, 2007, pp. 31-36.
6. M. Utting, A. Pretschner, B. Legeard, “A taxonomy of model-based testing approaches,” *Journal of Software Testing, Verification and Reliability*, 2012;, 22(5), pp. 297-312.
7. S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, B.M. Horowitz, “Model-based testing in practice,” in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 285-294.
8. A. van Lamsweerde, “Formal specification: A roadmap,” in *Proceedings of the Conference on the*

Future of Software Engineering, ACM Press, 2000, pp. 147–159.
9. C.B. Jones, *Systematic Software Development Using VDM*, Second Edition, Prentice Hall, 1990.
10. J.R. Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
11. ISO/IEC, *Z Formal Specification Notation - Syntax, Type System and Semantics*, First Edition, ISO/IEC 13568, Information Technology, 2002.
12. G.T. Leavens, A.L. Baker, C. Ruby, "JML: A notation for detailed design," *Behavioral Specifications of Businesses and Systems*, 1999, pp. 175-188.
13. Object Management Group, *OMG Unified Modeling Language*, Version 2.5, 2012, http://www.omg.org/spec/UML/2.5/Beta1/PDF.
14. Object Management Group, *Object Constraint Language Specification*, Version 2.0, 2006.
15. H. Simonis, "Applications of constraint logic programming," in *Proceedings of the Twelfth International Conference on Logic Programming*, 1995, pp. 9-11.
16. V. Kumar, "Algorithms for constraint satisfaction problems: A survey," AI Magazine, 1992, 13(1), pp. 32-44.
17. K.R. Apt, M.G. Wallace, *Constraint Logic Programming Using ECLiPSe*, Cambridge University Press, 2007.
18. K. Beck, E. Gamma, *JUnit Cookbook*, http://junit.sourceforge.net/.
19. P. Sommerlad, *CUTE - C++ Unit Testing Easier*, 2011, http://cute-test.com.
20. Object Technology International Incorporation, *Eclipse Platform Technical Overview*, 2003.
21. R. A. DiMillo, A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, 1991, 17(9), pp. 900-910.
22. T. A. Budd, D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, 1982, 18(1), pp. 31-45.
23. L. J. Morell, "Theoretical insights into fault-based testing," in *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, 1988, pp. 45-62.
24. B. K. Aichernig, P. A. Pari Salas, "Test case generation by OCL mutation and constraint solving," in *Proceedings of the Fifth International Conference on Quality Software*, 2005, pp. 64-71.
25. A. Gotlieb, B. Botella, M. Rueher, "Automatic test data generation using constraint solving techniques," in *Proceedings of the 1998 ACM/SIGSOFT Symposium on Software Testing and Analysis*, 1998, pp. 53–62.
26. A. Gotlieb, B. Botella, M. Rueher, "Symbolic execution of floating-point computations," *Journal of Software Testing, Verification and Reliability*, 2006, 16(2):97-121.
27. A. Gotlieb, B. Botella, M. Rueher, "A CLP framework for computing structural test data," in *Proceedings of the First International Conference on Computational Logic*, 2000, pp. 399–413.
28. C. Meudec, "ATGen: Automatic test data generation using constraint logic programming and symbolic execution," *Journal of Software Testing, Verification and Reliability*, 2001, 11(2):81-96.
29. C. Lembeck, R. Caballero, R. A. Muller, and H. Kuchen, "Constraint solving for generating glass-box test cases," in *Proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming*, 2004, pp. 19–32.
30. N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: automatic generation of path tests by combining static and dynamic analysis," in *Proceedings of the 5th European Dependable Computing Conference*, 2005, pp. 281–292.
31. K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005, pp. 263–272.
32. A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid, "Korat: a tool for generating structurally complex test inputs," in *Proceedings of the 29th international conference on Software Engineering*, 2007, pp. 771–774.
33. V. Senni and F. Fioravanti, "Generation of test data structures using constraint logic programming," in *Proceedings of the 6th international conference on Tests and Proofs*, 2012, pp. 115–131.
34. J. Dick, A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications," in *Proceedings of the 1st International Symposium on Formal Methods Europe on Industrial-Strength Formal Methods*, 1993, pp. 268–284.
35. H. Lotzbeyer, A. Pretschner, E. Pretschner, "AutoFocus on constraint logic programming," in *Proceedings of (Constraint) Logic Programming and Software Engineering*, 2000.
36. A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, "Model based testing for real: The inhouse

card case study," *International Journal on Software Tools for Technology Transfer*, 2004, 5(2-3), pp. 140-157.

37. F. Bouquet, F. Dadeau, B. Legeard, "Using constraint logic programming for the symbolic animation of formal models," in *Proceedings of the International Workshop on Constraints in Formal Verification*, Tallinn, Estonia, 2005, pp. 32-46.
38. S. Colin, B. Legeard, F. Peureux, "Preamble computation in automated test case generation using constraint logic programming," *Journal of Software Testing, Verification and Reliability*, 2004, 14(3), pp. 213-235.
39. Swedish Institute of Computer Sciences, *SICStus Prolog 3.8.7 Manual Documents*, October 2001, http://www.sics.se/sicstus.html.
40. F. Bouquet, B. Legeard, F. Peureux, "CLPS-B: A constraint solver to animate a B specification," *International Journal on Software Tools for Technology Transfer*, STTT, 2004, 6(2), pp. 143-157.