Automated Test Data Generation Using a Relational Approach

Insang Chung
Hansung University
Depart. of Computer Eng., Seoul, Korea
insang@hansung.ac.kr

James M. Bieman
Colorado State University
Depart. of Computer Sci., Ft. Collins CO 80523
bieman@cs.colostate.edu

Abstract

In general, test data generation techniques require an entire program path for automated test data generation. This paper presents a new way for generating test data automatically even without specifying a program path completely. The proposed method reduces the burden of selecting a program path and also makes it easy to generate test data according to various test adequacy criteria. For the ends, this paper presents a framework for transforming a program under test into Alloy which is the first-order relational logic and then producing test data via Alloy analyzer. This paper illustrates the proposed method through simple, but illustrative examples.

Keywords: *Program Testing, Alloy, Test Data Generation.*

1. Introduction

Test data generation techniques can be divided into two categories: path-oriented and goal-oriented approaches[1]. The path-oriented approach requires a program path and identifies input values which will traverse the path. This approach forces us to select a path for each uncovered statement to satisfy a structural coverage criteria like statement coverage. It can also waste time and resource in the case where no input values exist for the given path.

The goal-oriented approach generates test data which will exercise a specific program point rather than a program path. It provides us with a possibility of selecting among a set of paths as far as they reach the target. Many goal-oriented techniques require the actual execution of a program, which means that we can make use of run-time information in order to compute test data more accurately[2]. It, however, has some technical difficulties in fulfilling some test adequacy criteria such as branch or data flow coverage criteria which require more than one program points. For example, data flow coverage requires two program points. One is a program point which defines the variable and the other is to use the variable. To cope with those coverage criteria, the goal-oriented approach needs to be extended to accept two or more target points.

This paper presents a technique which can generate test data for a specified sequence of program points. That is, one can give one or more specific program points or even an entire program path for test data generation. This is achieved through a translation of the program under test into Alloy which is a formal modelling language based on the first-order relational logic[3]. This paper will illustrate how to transform a program into Alloy and how to generate test data according to various test adequacy criteria.

The rest of the paper is organized as follows. The next section will give a brief overview of Alloy. Section 3 will present the translation rules of a program into Allot for automated test data generation. In Section 4, we will discuss future work.

2. Alloy

Alloy is a formal modelling language based on the first-order relational logic. An Alloy specification is a sequence of two kinds of paragraphs: signatures that are used for defining new types and formula paragraphs such as facts and predicates to record constraints.

As an example, we declare two signatures, namely `People` and `Fish`:

```
sig People{ catch : set Fish }    sig Fish{}
```

Each signature introduces a basic type which denotes a set of atoms. These atoms can be mapped by the relations declared in the signatures. In the declaration of `People` the field `catch` denotes the relation between `People` and `Fish` and the `set` relation qualifier specifies that `catch` maps each atom in `People` to a set of atoms in `Fish`. This indicates that a person can catch many fishes.

In Alloy, signatures can extend a signature, establishing that the extended signatures are just subsets which are disjoint from each other. For example,

```
sig CatFish, Muller extends Fish{}
```

This example shows that `CatFish` and `Mullet` denote subsets of `Fish` which do not have objects in common.

A signature can be *abstract* to hold only those elements that belong to one of those signatures that extend it. That is, signatures that extend an abstract signature partition it. In order to designate a signature as an abstract signature, the `abstract` keyword is used before the declaration of the signature.

```
int mid(int x, int y, int z) {
  int mid;
  mid=z;
  if (y<z) {
    if (x<y)  mid=y;
    else if (x<z) mid=x;
  } else {
    if (x>=y) mid=y;
    else if (x>z) mid=x;
  }
}
```

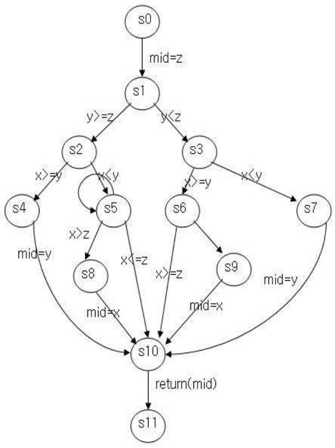

Figure 1. An example program and the finite state machine

Formula paragraphs include facts and predicates. Facts are used to represent invariants which always hold. The following example introduces a fact stating that “a fish can be caught by at most one person.”:

```
fact {
        all f:Fish|lone p:People|f in p.catch
}
```

The operator `in` denotes a subset or membership operator and the dot operator represents the relational composition. Since `p` is a singleton set which can be regarded as a unary relation, `p.catch` represents the functional image of `p` under `catch`.

The keyword `pred` is used to declare a predicate which evaluates to either true or false. For example, the following predicate named `doFishing` states that each person catches at least one fish:

```
pred {
    all p:People|some f:  Fish|    p → f in catch
}
```

Here, p→f corresponds to a tuple consisting of `p` and `f`, i.e., `(p, f)`. Suppose that we want to find a model of the predicate (that is, an assignment of values to the sets and relations that make the predicate true). This can be done by:

```
run doFishing for exactly 3 People, 4 Fish
```

This has the Alloy analyzer find a model that will make the predicate true by using exactly 3 atoms for People and at most 4 atoms for Fish.

3. Encoding Programs in Alloy

The proposed method first derives a computation graph from the code and transforms the graph into the finite state machine. We will present how to encode the finite state machine in Alloy for test data generation.

3.1. Deriving the computation graph

A computation graph is essentially a control flow graph if the program does not have loops[4]. A node represents a program control point and an edge represents either a predicate test or an elementary statement such as an assignment. If there exist loops in the program, the computation graph is a control flow graph which unrolls the loops finite times. There are no cycles in computation graphs. For example, if we apply one rolling to the following program fragment:

```
a; while (p) s; b
```

results in the computation graph which would be the same as the control flow graph of the program:

```
a; if (p) s; b
```

3.2. Constructing the finite state machine

A finite state machine(FSM) has to be constructed in such a way that we can find input which will execute a specified set of program nodes in a certain order even if we do not give the specification of a concrete path between each node. For clarity, we will refer to the set of nodes to be covered as *essential nodes*. We will also refer to an essential node as a *target node* if it is the last node in the execution sequence. It is straightforward to construct a finite state machine from a computation graph:

- each node in the computation graph corresponds to a state in the FSM.
- each edge in the computation graph corresponds to a transition in the FSM.
- a predicate test or a statement on each edge is translated into an Alloy formula which will serve as a condition that have to be fulfilled to enable the transition corresponding to the edge. In the next section, we will discuss how to transform a predicate test or an assignment statement into Alloy.

- a state corresponding to the target node has a self-transition. We will call the state the target state.

Fig. 1 shows that an example program and the finite state machine constructed from its computation graph before translating the predicate tests and the assignment statements to Alloy formulas. Note that state `S5` has the self-transition, which indicates that we have a test requirement that the program point corresponding to `S5` has to be executed. If we have another program points, i.e., essential nodes to be executed prior to `S5`, then we need to annotate the finite state machine with them.

3.3. Transforming a FSM into Alloy

We transform predicate tests and assignments statements into Alloy formulas based on SSA(Static Single Assignment) form[5]. A key property of SSA form is that each variable has a unique static definition point. This property allows us to deal with program variables as logical variables. As a result, all statements or predicates tests in a program can be dealt with as equalities or inequalities.

```
abstract sig Integer { val: Int }
sig X, Y, Z, MID extends Integer {}
```

Figure 2. Variable modelling

We define a signature for each program variable and use a new atom in the signature whenever the variable is defined so that each variable should have at most one definition. Fig. 2 shows the declarations of signatures for the program variables in the program of Fig. 1. The field `val` is introduced to represent an value that program variables of integer type hold.

```
abstract sig State {
     x: one X,
     y: one Y,
     z: one Z,
     mid: one MID
}

S0, S1, S2, ..., S11 extends State {}
```

Figure 3. State modelling

Fig. 3 shows how to represent the program state. Each program state represents the variable instances through the fields `x`, `y`, `z` , and `mid`. For example, `int s.x.val=10` [1] asserts that the variable x has the integer value 10 in state `s`.

Predicate tests and assignments of the code have to be transformed to Alloy formulas to describe state transitions. We illustrate the transformation rules through an example.

[1]The `int` operator is applied to the expression `s.x.val` to get the primary integer value associated with the expression.

```
pred doAssign1(s, s': State) {
  some mid': MID | s.mid !=mid' && mid'=s'.mid
  int s'.mid.val=s.z.val
  s'.x=s.x
  s'.y=s.y
  s'.z=s.z
}
```

Figure 4. An example to transform the assignment statement "`mid=z`" into an Alloy formula

Fig. 4 gives an example translation of the assignment `mid=z` of the program in Fig. 1. This formula accepts two states, `s` and `s'`. The state `s` represents the pre-state before the assignment and the state `s'` represents the post-state after the assignment. The first part of the formula asserts that the instances of the program variable `mid` at `s'` and `s` should be different. Since the assignment updates the `mid` variable, a new instance of `mid` needs to be introduced. The second part equates the value of `mid` at `s'` with the value of `z` at `s`. This describes the effect of the assignment. The rest of the formula is about frame conditions saying that all other variable instances except `mid` do not change. On the other hand, we have to add frame conditions for all variables in the case of the transformation of predicate tests because no variables are not changed by the predicate test.

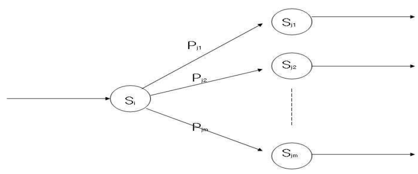

Figure 5. An example program and the finite state machine

We are now in a position to define the transition rule from a state to the next. Let us consider Fig. 5. If the current state is $\mathtt{S_i}$, then we can make a transition to one of the states $\mathtt{S_{jk}}$ (for k$\in$ {1,..,m}). Of course, $\mathtt{P_{jk}}$ should be fulfilled when the transition to $\mathtt{S_{jk}}$ is made. Thus, we have:

$$\mathtt{s\ in\ S_i} \Rightarrow \mathtt{P_{i1}(s,s')\&\&s'\ in\ S_{j1}} || \cdots || \mathtt{P_{im}(s,s')\&\&s'\ in\ S_{jm}}$$

If $\mathtt{S_i}$ has a self-transition (i.e., $\mathtt{S_i}$ is a state corresponding to the target node), $\mathtt{P_{ii}}$ would be the predicate `SKIP` defined as follows:

$$
\begin{array}{l}
\mathtt{pred\ SKIP(s,\ s':\ State)}\ \{ \\
\quad \mathtt{s'.x_1 = s.x_1} \\
\quad\quad \vdots \\
\quad \mathtt{s'.x_n = s.x_n} \\
\}
\end{array}
$$

SKIP represents frame conditions stating that the self-transition should not cause any change to all program variables.

```
open util/ordering[State] as so // import ordering utility
fact doTrans  {
  so/first() in S0
  all s:  State-so/last() |
    let s'=so/next(s) |
      s in S0 ⇒ doAssign1(s,s')&&s' in S1
      s in S1 ⇒ Comp1(s,s')&&s' in S2
                ||Comp2(s,s')&&s' in S3
      s in S2 ⇒ ···
            ⋮
      s in S5 ⇒ Comp5(s,s')&&s' in S8
                ||Comp6(s,s')&&s' in S10
                ||SKIP(s,s')&&s' in S5
      s in S6 ⇒ ···
            ⋮
}
```

Figure 6. Transition rules

Fig. 6 shows the transition rules for the FSM in Fig. 1. In the transition rules, the states are constrained to be ordered and the formulas such as doAssign1, Comp1, Comp2, etc are the translations of the program statements corresponding to each transition into Alloy. In addition, the constraint on the initial state is given.

3.4. Test data generation

If we have a sequence $< S_1, \cdots, S_k >$ to be covered, then we just need to designate S_k as the target state and the rest of the states as the essential states. This can be done by the formula:

```
pred testIt(s, s':  State)   {
 so/last() in Sk
 some S1
 ...
 some Sk-1
}
```

Suppose that we want to generate input which will cause the traversal of the assignment mid=y on the transition from S_4 to S_{10} in the FSM of Fig. 1. This can be simply done by finding an instance that will the following formula true:

```
pred testIt(s, s':  State)   {
 so/last() in S10
 some in S5
}
```

This example shows how to generate test data according to the statement coverage criterion. It is straightforward to apply our approach to the branch coverage criterion because each branch can be specified by a pair of two program nodes as in the case of the statement coverage criterion. Table 1 shows test data produced by the Alloy analyzer which satisfy the branch coverage criterion for the program in Fig. 1. The Alloy analyzer also produces program paths to be executed by the test data.

Table 1. Test data satisfying the branch coverage criterion

Target branch	Test Data $< x, y, z >$	Program path	Tested branches
$< S_2, S_4 >$	$< 2, 2, 0 >$	$< S_0, S_1, S_2, S_4 >$	$< S_1, S_2 >$ $< S_2, S_4 >$
$< S_5, S_8 >$	$< 2, 3, -3 >$	$< S_0, S_1, S_2, S_6, S_8 >$	$< S_1, S_2 >$ $< S_2, S_5 >$ $< S_5, S_8 >$
$< S_5, S_{10} >$	$< -4, 2, 2 >$	$< S_0, S_1, S_2, S_5, S_{10} >$	$< S_1, S_2 >$ $< S_2, S_5 >$ $< S_5, S_{10} >$
$< S_6, S_{10} >$	$< 3, -4, -3 >$	$< S_0, S_1, S_3, S_6, S_{10} >$	$< S_1, S_3 >$ $< S_3, S_6 >$ $< S_6, S_{10} >$
$< S_6, S_9 >$	$< 0, 0, 1 >$	$< S_0, S_1, S_3, S_6, S_9 >$	$< S_1, S_3 >$ $< S_3, S_6 >$ $< S_6, S_9 >$
$< S_3, S_7 >$	$< -4, 0, 2 >$	$< S_0, S_1, S_3, S_7 >$	$< S_1, S_3 >$ $< S_3, S_7 >$

4. Conclusion

While our approach offers improvements in complementing weak sides of path-oriented and goal-oriented techniques, there are some issues that are worthy of further research. First, support for inter-procedural test generation is needed. Even though it is still true that most of techniques for automated test data generation focus on unit level, it is necessary to extend our approach to inter-procedural level for more effective testing. Second, tools to support our technique have to be developed in order to put it in practice. Finally, we need to extend our work to support various programming language features including composite data types and pointers.

References

[1] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verfication and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[2] F. Roger and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering Methodology*, vol. 5, no. 1, pp. 63–86, 1996.

[3] D. Jackson, "Automating first-order relational logic," in *Proc. ACM SIGSOFT Conf. Foundations of Software Eng.*, 2000.

[4] D. Jackson and M. Varziri, "Finding bugs with a constraint solver," in *Proc. International Conf. on Softare Testing and Analysis*, 2000.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and System*, vol. 13, no. 4, pp. 451–490, 1991.