General Predicate Testing

Kovács Attila

Professor, Ph.D.

Andi Péter

Computer Science MSc

Budapest, 2023

Contents

1 Introduction

1.1 Common testing practices

1.2 Test case generation

2 Other test case generation methods

2.1 Equivalence Partitioning

2.2 Boundary Value Analysis

2.3 Random testing

3 Intervals

Intervals are the backbone of GPT. Instead of assigning single point to Equivalence Partitions, we use intervals. This way we can represent every possible value that we want to test our predicates with.

There weren't any off-the-shelf libraries that implemented interval handling exactly in the way I needed, so I had to create my own interval library. It consists of two main parts: Simple intervals and what I call Multiintervals. Multiintervals are made up of multiple simple intervals, but behave as an interval.

I will use the word interval both for simple intervals, or multiintervals. If a distinction needs to be made, I'll clarify. But in later chapters all that'll matter is that we are working with an interval.

This interval implementation has three important functions: intersection, union, and complement. With these I could implement all of GPT's functionality.

TODO: There was one Rust lib that had pretty good single intervals

3.1 Simple Intervals

Simple intervals are intervals in a mathematical sense. It has two endpoints, a lo (low) and a hi (high). It has two boundaries: a lo boundary and a hi boundary. A boundary is either closed [ ] and open ( ).

A special case is when an interval is unbounded. I'll use the notation of ∞ at and endpoint to mean that that side of the interval is unbounded, like: $[10, \infty)$.

An interval can contain only a single point: $[10, 10]$.

Empty intervals can exist when no points can be in an interval, like $(10, 10)$ or $(10, 10]$

An interval can only be constructed if lo $\leq$ hi.

For the implementation, I'm only sotring the lo_boundary, lo, hi, hi_boundary variables. All the calculations are made with these values.

TODO: A design alternative could be to represent all the possible states interval could be in in an Algebraic Data Type, so we cannot create inconsistent state, like $[-\infty, \infty]$ or $(10, 10]$

3.1.1 Intersection

Can intersect?

Pseudocode for detecting whether the intersection of two intervals is possible:

```
case1 = self.lo > other.hi || other.lo > self.hi
case2 = self.lo == other.hi && (self.lo_boundary == Open ||
other.hi_boundary == Open)
case3 = other.lo == self.hi && (other.lo_boundary == Open ||
self.hi_boundary == Open)

return !(case1 || case2 || case3)
```

Here, we are looking at if the intervals don't intersect, because those are the easier cases to handle. We can negate this result to get when intervals intersect.

`case1` is when the intervals are above or below each other, like $[0, 10]$ and $[20, 30]$ or $[20, 30]$ and $[0, 10]$. Because lo $\leq$ hi we only have to check the lo and hi of the two intervals.

`case2` is when the intervals would have the same endpoint, but either one of the boundaries is Open. Because an open boundary doesn't contain that point, intersection can't be made as well. For example $[0, 10]$ and $(10, 20)$.

`case3` is the same as `case2`, but checking the intervals from the other way.

In all other cases the intervals would intersect.

Calculating the intersection

Pseudocode for calculating the intersection of two intervals `self` and `other`:

TODO: Here we need a lo_cmp and a hi_cmp instead of the > or <, because if they have the same value the boundary makes the difference. Also, when comparing intervals (for intersection, union, or complement) it would make the cases more explicit.

```
if self doesn't intersects_with other:
    return No Intersection

interval_with_lower_hi = if self.hi > other.hi then self else other
interval_with_higher_lo = if self.lo < other.lo then self else other

return Interval {
    lo_boundary: interval_with_higher_lo.lo_boundary
    lo: interval_with_higher_lo.lo
    hi: interval_with_lower_hi.hi
    hi_boundary: interval_with_lower_hi.hi_boundary
}
```

First we check if the intervals can even be intersected. If they don't, we return that there is no intersection. In Rust this is an `Option::None` type, in other languages it might be a `null`.

Then, if the intervals can be intersected, we look for the hi endpoint which is lower and use that as the hi point and hi boundary. We do the same and look for higher lo point and use that as a lo point.

3.1.2 Union

The union of two simple intervals can either be a simple interval (if they intersect) or a multiinterval if they don't. Because of this, we always assume that the union of two simple intervals will be a multiinterval. There is no need for the code to produce a union of simple intervals that is a simple interval, but it could be implemented.

Pseudocode:

```
return Multiinterval::from_intervals([self, other])
```

We can just create a multiinterval from the two intervals. With this constructor Multiinterval will call a `clean()` on itself, which I'll explain later. In short, in this case `clean()` would merge the two intervals if they intersect.

3.1.3 Complement

TODO: There is no inverse defined for Simple intervals lol. (I only needed complements for Multiintervals and that doesn't need the complements of simple intervals to work)

3.2 Multiintervals

Multiintervals are intervals composed of multiple simple intervals. They are required for GPT, because for example if we have a predicate that states that $x > 0 \land x \leq 10 \land x \notin \{5, 7\}$ we could represent the interval of values x could take as the the multiinterval $(0, 5)$ $(5, 7)$ $(7, 10]$.

An empty multiinterval is one which has no intervals. We don't store empty intervals in mutliintervals.

An invariant of the Multiinterval is that its intervals are sorted in increasing order.

3.2.1 Cleaning

There could be multiintervals, which are not 'clean', or not in a sematically correct form. Take for example $(10, \infty)$ $(-5, 5)$ $[0, 20]$.

Here are the steps to clean a multiinterval:

1. **Removing empty intervals.** Empty intervals hold no values, so they are unnecessary to have in a multiinterval. From the example we'd remove $(-5, -5)$

2. **Sorting the intervals.** Intervals should be in an increasing order inside a multiinterval. When comparing intervals we compair their los. The example would change from $(10, \infty)$ $[0, 20]$ to $[0, 20]$ $(10, \infty)$.

3. **Merging overlapping intervals.** If intervals would would intersect, we can merge them together. The example would become from $[0, 20]$ $(10, \infty)$ to $[0, \infty)$.

We can define a constructor `Multiinterval::from_intervals` that will always create a clean multiinterval from a list of intervals:

```
def Multiinterval::from_intervals(intervals):
   multiinterval = new Multiinterval(intervals)

   multiinterval.clean()

   return multiinterval
```

3.2.2 Intersection

Can intersect? To check that two mutliintervals intersect, we can check if any of their intervals intersect. Pseudocode:

```
for x in self.intervals:
   for y in other.intervals:
      if x.intersects_with(y):
         return True

return false
```

TODO: Because the intervals are in increasing order, we could do an $O(n)$ algorithm instead of an $O(n^2)$

Calculating the intersection

Pseudocode:

```
intersected_intervals = []

for x in self.intervals:
   for y in other.intervals:
      if x.intersects_with(y):
         intersected_intervals += x.intersect(y)

return Multiinterval::from_intervals(intersected_intervals)
```

We try to intersect all the intervals. The `Multiinterval::from_intervals` constructor will call a `clean()`, so the resulting intersected Multiinterval will be in the correct form.

3.2.3 Union

We take both Multiintervals' intervals, concatenate them and create a Multiintevral from them. This constructor call `clean()`, so it'll take care of sorting and overlapping intervals.

```
return Multiinterval::from_intervals(self.intervals ++ other.intervals)
```

3.2.4 Complement

The complement of an interval contains all the elements which are not in the interval. In simple terms, we just return all the 'space' between our intervals.

For example:

- Multiinterval: $[-42, 3)\ (3, 67)\ (100, 101)\ [205, 607]\ (700, \infty)$
- Complement: $(-\infty, -42)\ [3, 3]\ [67, 100]\ [101, 205)\ (607, 700]$.

Pseudocode:

```python
if self.is_empty()
    return (-infinity, infinity)

complement_intervals = []

if intervals.lowest_lo() != -infinity:
    complement_intervals += Interval(Open, -infinity, lowest_lo,
lowest_boundary)

for [a, b] in self.intervals.window(2):
    complement_intervals += Interval(a.hi_boundary.complement(), a.hi,
b.lo, b.lo_boundary.complement())

if intervals.highest_hi() != -infinity:
    complement_intervals += Interval(highest_boundary, highest_hi,
infinity, Open)

return new Multiinterval(complement_intervals)
```

We go through the intervals in pairs. The `.window(2)` function returns all the neighbouring pairs in a list. It is a sliding window. We always create an interval between the hi of the first and the lo of the second interval, as this is the space not covered by our multiinterval.

As in the sliding window we only look at the hi of the left side and the lo of the right side, we have to handle the case at the edges of the multiinterval. If our multiinterval is not unbounded, the complement has to be unbounded.

We don't have to clean the Multiinterval, because of its invariants it won't have empty intervals, it will be sorted, and it won't have overlapping intervals.

4 GPT Algorithm

TODO: Explain precision **TODO:** In this doc when we refer to intervals, we could mean multiintervals, for the sake of simplicity.

4.1 BVA in GPT (in, inin, on, off out)

4.1.1 Converting conditions to intervals

In GPT we create intervals from conditions, similar to Equivalence Partitioning.

Example: For the condition $x < 10$, we create the interval $(-\infty, 10)$

TODO: Explain the condition -> to interval process

An interesting case happens, when we take the not equal to condition. For $x \neq 10$ we have to generate a multiinterval, because the values x could take is $(-\infty, 10)$ $(10, \infty)$.

4.1.2 Extending BVA

Now that we have intervals to work with, we should generate the possible test values. In normal BVA we would pick single points from the intervals. GPT extends this in two ways:

1. We don't pick single values, but the the largest possible intervals. This will be helpful, when in the end we try to reduce the number of test cases, we can see the overlap between different test cases.
2. We not only look at the minimum and maximum possible acceptable value, but more. We look at not acceptable values and test that our implementation properly fails for those. **TODO:** This might be covered in BVA, further investigation needed.

TODO: Actually, GPT is an implementation of BVA with some extra functionalities, right?

4.1.3 Equivalence Partitioning in GPT

In BVA the equivalence partitions for $[1, 10)$ would be $(-\infty, 1)$ $[1, 10)$ $[10, \infty)$

In GPT we have more equivalence partitions. **TODO:** Explain why

- **In:** The interval of the acceptable values. In case of Open ends we step one with the precision to make it closed. Unbounded parts remain unbounded.

Example: $[1, 10)$ will have the In of $[1, 9.99]$
Example: $(-\infty, 10]$ will have the In of $(-\infty, 10]$

- **InIn:** One step of precision inside In.

Example: $[1, 10)$ will have the InIn of $[1.01, 9.98]$
Example: $(-\infty, 10]$ will have the InIn of $(-\infty, 9.99]$

- **On:** First possible acceptable values from the edges. These are single points, the endpoints of In. Unbounded edges have no On points. If the interval is a single point there will only be one On point.

Example: $[1, 10)$ will have the On points of $[1, 1]$ $[9.99, 9.99]$
Example: $(-\infty, 10]$ will have the On point of $[9.99, 9.99]$

- **Off:** First not acceptable values from the edges. One step outside the edges of In. Unbounded edges have no Off points.

Example: $[1, 10)$ will have the Off points of $[0.99, 0.99]$ $[10, 10]$
Example: $(-\infty, 10]$ will have the Off point of $[10.01, 10.01]$

- **Out:** Not acceptable values, except for the Off points. The complement of the In interval, stepped one, to exclude the Off points.

Example: $[1, 10)$ will have the Out interval of $(-\infty, 0.98]$ $[10.01, \infty)$
Example: $(-\infty, 10]$ will have the Off interval of $[10.01, \infty)$

TODO: Here the In interval and On points are overlapping, why the "duplication"? Because in GPT when we create the Off and Out intervals we only do it for one variable in a test case. That way we only test that that variable is handled correctly in the logic. All the other variables will have the In intervals, so they are accepted.

For multiintervals we use the same equivalence partitioning technique. We calculate the partition for all the intervals inside the multiinterval and create a multiinterval out of the partitions.

4.1.4 Test case generation in GPT

4.2 Hierarchical GPT

5 GPT Lang

5.1 Parser

5.2 AST, IR

6 Graph Reduction

6.1 Why even graph reduce

6.2 MONKE

6.3 Least Losing Nodes

6.4 Least Losing Edges

7 Code architecture

7.1 Rust

7.2 Frontend app, webassembly

8 Future improvement ideas

Bibliography