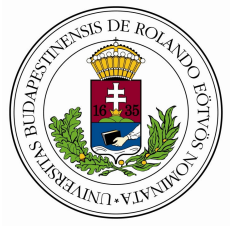

Eötvös Loránd University
Faculty of Informatics
Dept. of Computer Algebra

General Predicate Testing

Kovács Attila
Professor, Ph.D.

Andi Péter
Computer Science MSc

Budapest, 2023

Contents

Abstract

Software testing is a critical part of the Software Development Life Cycle. Test designers often have to create test cases manually from the requirements, which is costly and prone to human error. This is partly because natural language requirements are hard to formalize, and there are currently no tools to do it automatically.

Boundary Value Analysis (BVA) is a powerful black-box method for testing programs. In their book Kovács Attila and Forgács István proposed General Predicate Testing (GPT), which is a systemic test case generation method based on BVA. There have been some automatic test generation tools based on BVA, but none on GPT. The book describes the GPT method for test designers for manual application, but that can be prone to human error. The GPT method was only defined for conjunctive forms, so if the requirements had disjunctions, the test designers had to convert them to conjunctions by hand. Furthermore, there was no formal way of reducing the number of test cases by hand, it relied on intuition.

In this thesis I'll present an algorithmic formalization of the GPT method, making it possible to be easily implemented and integrated into automatic test generation tools. Based on that, I've created an implementation and published it as an open-source tool. I've created a domain specific language (DSL) for specifying natural language requirements in a way that can be used for GPT. In addition, I added support to express disjunctions in the DSL. I've also defined an algorithm to reduce the number of initially generated test cases significantly. In some cases the reduction can reach 90.32%, while still covering all initial test cases. With this tool, the amount of time spent by test designers on General Predicate Testing can be significantly reduced.

Terminology

ADT: Algebraic Data Type
AST: Abstract Syntax Tree
BVA: Boundary Value Analysis
CPH: Competent Programmer Hypothesis
DSL: Domain Specific Language
EP: Equivalence Partitioning
GPT: General Predicate Testing
IR: Intermediary Representation
LLC: Least Losing Components
LLE: Least Losing Edges
LLNR: Least Losing Nodes Reachable
MLE: Most Losing Edges
MONKE: Minimal Overhead Node Kollision Elliminator
SUT: System Under Test

1 Introduction

1.1 Motivation

Software plays an ever-increasing role today: From entertainment, and household appliances to self-driving cars, or medical devices. It is apparent, that in critical systems the margin of error is zero. There have been some examples, where simple errors caused catastrophic failures. For example, a NASA orbiter had a metric conversion error, which caused the loss of the $125 million orbiter [1].

Software testing is an essential part of the Software Development Life Cycle [2]. Testing software allows us to be confident that the program adheres to the requirements and works as expected. There can be functional and non-functional requirements, which our systems have to satisfy. In this thesis I'll focus on functional requirements.

There are many testing strategies and tools to choose from [3]. Because we're trying to catch errors in the implemented programs, we should not have erroneous tests. There are tools which can generate test cases automatically [4]. These are advantageous, because they reduce the risk of human error and are usually faster than manual test case generation.

Shipping software without comprehensive tests is a shortcut companies can make, when they don't want to spend too much time on testing and want to release as early as possible. If there were ways that'd make comprehensive testing accessible and would take only a small amount time, it'd solidify the culture around testing. This is why, I think, researching the field of automatic test generation tools is valuable.

1.2 Common testing practices

There are multiple approaches to testing software, one is black-box testing, where we create tests sets from the requirement specifications. In practice, this means that we're not looking at how to code is written when writing tests. This way, we can systematically test the correctness of outputs for given inputs [5].

The state-of-the-art black-box testing methods are: [6] [3] [2]

1. *Equivalence Partitioning*: The input and output domains can be partitioned in a way, that values in each partition belong to the same Equivalence Class. This way, test cases are only required to have one value from each partition.
2. *Boundary Value Analysis*: Test cases are created from the boundaries of Equivalence Classes. These can be the values just below, on, or just above the boundaries. This can catch usual off-by-one errors.
3. *Fuzzing*: Black-box fuzz testing is about taking valid inputs and randomly mutating them to try to find implementation bugs. This approach has low code-coverage and requires a lot of test cases. [7] There is also white-box fuzzing, which is much more effective, due to having access to the source code [8].

4. *Cause-Effect Graph*: We create a graph and creating links between the effect and its causes. There are four types of these links: identity, negation, logical OR, and logical AND. There are some proposed automatic test generation tools from Cause-Effect Graphs [9].
5. *Orthogonal Array Testing*: OAT is a pairwise testing technique used when the input domain is small, but testing all the possible combinations of inputs would result in a too large test set [10].
6. *All Pair Testing*: All the unique pairs of inputs are in the test case set. This way all the possible pairs are tested, but the test set is quite large.
7. *State Transition Testing*: Used for state machines or User Interfaces, where the transitions between states are tested.

In this thesis I'll assume, that the input variables are independent. Otherwise, domain analysis has to be used [11] [12] [13].

Next, I'll explain Equivalence Partitioning and Boundary Value Analysis in more detail, as GPT is based on these methods.

1.2.1 Equivalence Partitioning

In Equivalence Partitioning, the inputs are divided into equivalence classes in a way, that if two inputs belong to the same class, they behave in the same way during testing. If both inputs test the same behavior, then if there is a bug, they can both detect it [2].

The equivalence classes are non-empty, disjoint, and the union of the equivalence classes cover the entire input domain. Equivalence classes are also referred to as partitions.

The partitions can be either valid or invalid partitions. Valid partitions contain the acceptable values, invalid partitions contain the not acceptable values. A value is acceptable if the predicate returns a logical true value.

The steps of Equivalence Partitioning are [2]:
1. Identify the input domain.
2. Partition the domain into valid and invalid.
3. Refine and merge the partitions until they can't be merged anymore.
4. Validate the partitioning.

Once we have the partitions, we can create test sets by creating a test case for each partition.

It is possible to obtain the partitioning data without actually doing the partitioning [2]. We can select the domain boundaries and use the boundaries to approximate the partitions. As borers are easier to compute than the entire partitions, and we can generate test cases from these borders, this is a good approximate solution for equivalence partitioning.

1.2.2 Boundary Value Analysis

In most cases, potential bugs occur near the border of equivalence partitions, because of errors made by programmers [2]. As such, we should select test cases from the partitions to test these boundaries.

Boundary Value Analaysis builds on Equivalence Partitioning and proposes ways to select test points from the partitions.

Forgács and Kovács state, that "many textbooks, blogs, software testing courses suggest inappropriate BVA solutions." [2, p. 74]. They propose the following method for selecting test values from equivalence partitions:

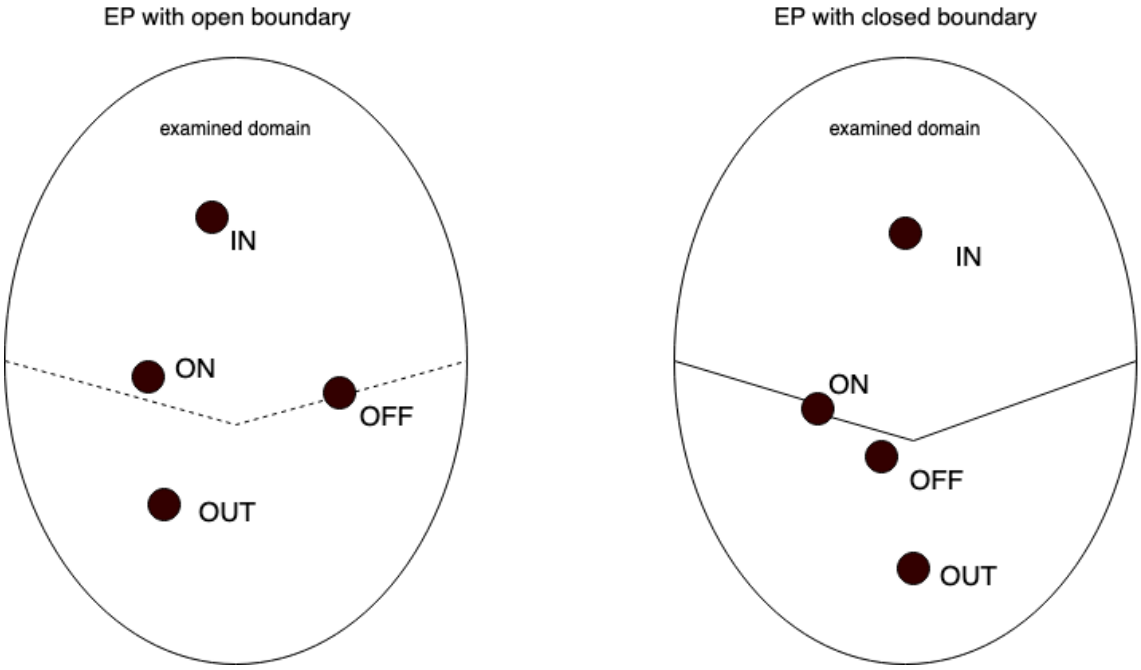

Figure 1: EP with closed or open boundaries [2, p. 75]

These IN, ON, OFF, OUT points will be important, as equivalence partitioning in GPT builds on this (detailed in Section 3.1.3)

Predicate errors: While programming, predicates can be written in many wrong ways. The numbers could be off-by-one [14], we could mistype the operator and have > instead of < or <=. Or we could write != instead of ==.

BVA helps to detect predicate errors, because these most often occur at borders of partitions. If we have correctly selected test cases to cover all the borders of partitions, we will correctly detect these predicate errors.

As Forgács and Kovács detailed, BVA is easy when we have one parameter, but once we have multiple parameters, it becomes significantly harder [2].

1.3 Competent Programmer Hypothesis

The Competent Programmer Hypothesis is: "Programmers create programs that are close to being correct, i.e., the specification and the appropriate code are close to each other." [2].

Let's look at an example. We have to create a function that tests if $x < 0$, where x is an integer. We can implement it as `x < 0` and have the test cases $-100, -1, 0, 1, 100$ to test it.

Let's say, that somebody implements this as `x < 5 && x != 4 && x != 3 && x != 2 && x != 1 && x != 0`. For integers, this implementation is correct, but it is needlessly complicated, and it is far from the specification. In this case, if the programmer makes a predicate error in `x < 5` and writes `x < 6`, the function will return `true` for 5, meaning `5 < 0`. Our original test cases didn't have this test point, so we wouldn't catch this bug.

It can be seen, that programs could be infinitely complicated this way, for example we could do $x < 100$ or $x < 99999999$ with the same principles. We can't test all potential predicate errors in every possible implementation. With CPH, we say that the implementation will be close to the specification, so it is enough to only test predicate errors, which would happen in an implementation close to the specification.

1.4 Automatic test case generation

Generating test cases automatically is advantageous, because creating test cases manually takes up a significant amount of time and is prone to human error.

White-box testing solutions usually have automatic test generation algorithms, as they can parse the source code and generate test cases from it [15] [8]. But if we generate test cases from a faulty System Under Test (SUT), one that runs, but doesn't adhere to the requirements, we might not catch those bugs. The usefulness of automatic white-box test generation was called into question by Gordon Fraser and Matt Staats, in their analysis they found that "there was no measurable improvement in the number of bugs actually found by developers" [16].

White-box test generation algorithms exist that use BVA [17], but these are different from black-box solutions.

Black-box testing solutions have a harder time generating test automatically, because they are derived from the specifications and requirements. Human language is hard to parse and even harder to extract test cases from [18]. With the recent advances of Generative Pre-trained Transformer models, maybe this will change, but current solutions are prone to so-called 'hallucinations' [19], and thus we can't rely on them to generate correct test cases.

To generate test cases automatically, black-box solutions have to use some kind of intermediary format to generate the tests from. Test designers have to convert the requirements to one of these formats. Most of these are formal specifications are in UML diagrams [4].

There are some black-box test generation methods using BVA, but they require the use of UML diagrams [20] [21]. These can't test simple functions, only classes with interfaces or state transitions.

BWDM [22] uses BVA to generate test cases automatically from the VDM++ specification language. In a followup paper they've used pair-wise testing to reduce the number of test cases generated by BVA [23].

1.5 Methodology

In this thesis, my main task was to create an implementation of the GPT algorithm. As the book only outlined how to do GPT manually, I had to come up with ways to make an automatic and algorithmic solution.

First, I had to implement proper interval handling, as there weren't any available libraries that handled intervals this way. In Section 2 I'll show how I implemented my own simple intervals and multiintervals.

Next, when I had working intervals, I had to implement the GPT algorithm. This went through a few iterations. In the first version I could create NTuples by hand and run the GPT test case generation algorithm on them, which generated a non-reduced set of test cases.

But creating NTuples by hand is quite some manual work, so I implemented a CSV-like structure for formalizing requirements. It was an easier and more user-friendly way to use GPT. I also created a web page for it that could be shared with other people.

Figure 2: CSV-like GPT Lang

This was a working minimal viable product (MVP), but it was hard to write and reason about requirements in this format. This is why I created GPT Lang, which I'll detail in Section 4. I had to design a domain specific language (DSL), that was easy to write and reason about, while also familiar to programmers, but still adhered to black box testing principles. I wrote a parser, designed an Abstract Syntax Tree (AST) and an Intermediary Representation (IR), and then transformed the IR to NTuples that GPT could use.

Because GPT Lang could resemble source code, one could think that we could extend it to analyze source code and generate test cases in a white box way. But as detailed in the next chapters, I explicitly wanted to avoid this, so GPT Lang is by-design only for black box testing.

After this, I had a user-friendly DSL for writing specifications in, but one challenge was, that the original GPT algorithm was only detailed for conjunctive forms. Disjunctions are an essential part of requirements and programming, so I wanted to research how I could make disjunctions work with GPT. In Section 4.5, I detail this procedure. This is a great lift for GPT, as now the test designers don't have to think about how to bring conjunctive forms to disjunctive forms, because my program would handle that automatically. It not only saves time, but reduces the risk of human error.

After that, I've researched how the number of test cases can be reduced in an algorithmic way. In Section 5, I detail how I abstracted this problem to be about graph reduction and what different graph reduction algorithms I came up with. Graph reduction is an essential part of GPT, because it reduces the number of test cases needed to test the same behavior by orders of magnitude. This was also a pretty hard procedure to do manually for GPT, so automation can save even more time for the test designers.

In Section 6, I'll validate that my implementation is correct and generates the test cases outlined in the book. This section also provides examples about how GPT works and what errors it can catch.

After the summary in Section 8, I'll talk about some future research and implementation ideas in Section 9.

2 Intervals

Intervals are the backbone of GPT. Instead of assigning single points 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. The `intervals-general` library [24] had good simple interval support, but no multiinterval support.

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 will matter is that we are working with an interval.

In my interval implementation there are three important functions: intersection, union, and complement. With these I could implement all of GPT's functionality.

2.1 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 [ ] or open ( ).

A special case is when an interval is unbounded. I'll use the notation of ∞ with an open boundary to indicate when an interval is unbounded from one side. For example: $[10, \infty)$.

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 storing the *lo_boundary*, *lo*, *hi*, *hi_boundary* variables. All the calculations are made with these values.

Interval could be in different states (unbounded, bounded, single point, empty). A design alternative could to use Algebraic Data Types to represent intervals in different states, so we cannot create inconsistent an state, like $[-\infty, \infty]$ or $(10, 10]$.

2.1.1 Intersection

Detecting if there is an intersection

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 the case when the intervals don't intersect, because that is an easier case to handle. We can negate this result to get whether 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. Since an open boundary doesn't contain that point, intersection can't be made either. 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 intervals `self` and `other`:

```
if not self.intersects_with(other):
   return "No Intersection"

interval_with_lower_hi = if self.hi_cmp(other) == Greater then self
else other
interval_with_higher_lo = if self.lo_cmp(other) == Less 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 can't, we return that there is no intersection. In Rust, this could be an `Option::None` type, in other languages it could be a `null`.

Then, if the intervals can be intersected, we take the lower *hi* endpoint and use that as the new *hi* point and *hi* boundary. We do the same and look for higher *lo* point and use that as the new *lo* point. The `hi_cmp` and `lo_cmp` functions are comparison functions, which take the boundaries into account when comparing *hi* or *lo*. These

functions are needed, because there could be a case where we the points are the same, but the boundaries are different. For example $5)$ and $5]$, in this case the closed boundary is higher.

2.1.2 Union

The union of two simple intervals can either be a simple interval (if they intersect) or a multiinterval (if they don't). Due to 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.

2.1.3 Complement

The complement of an interval contains all the elements which are not in the interval.

For simple intervals, this could result in a multiinterval. For example, the complement of $[0, 10)$ is $(-\infty, 0)$ $(10, \infty)$.

Simple intervals basically have two sides: a side left and a side right of the interval. If our interval is unbounded, we won't have that side in the complement. The complement of $(-\infty, \infty)$ is an empty multiinterval.

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


new_intervals = []

if self.lo != -infinity:
        new_intervals    +=    Interval(Open,    -infinity,    self.lo,
self.lo_boundary.inverse())


if self.hi != infinity:
      new_intervals += Interval(self.hi_boundary.inverse(), self.hi,
infinity, Open)

MultiInterval {
      intervals: new_intervals
}
```

2.2 Multiintervals

Multiintervals are intervals composed of multiple simple intervals. They are required for GPT. For example if we have a predicate that states $x > 0 \land x \leq 10 \land x \notin \{5, 7\}$ we could represent the interval of values x could take as 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 multiintervals is that its intervals are sorted in increasing order.

2.2.1 Cleaning

There could be multiintervals, which are not 'clean', i.e. not in a semantically 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 compare their *lo* values. The example would change from $(10, \infty)\ [0, 20]$ to $[0, 20]\ (10, \infty)$.
3. **Merging overlapping intervals.** If intervals would intersect, we can merge them together. The example would change 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
```

2.2.2 Intersection

Detecting if there is an intersection

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
```

As a future optimisation idea, we could do an $O(2n)$ algorithm instead of an $O(n^2)$, because the intervals are in increasing order. We could traverse them like in a two pointers technique [25].

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.

2.2.3 Union

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

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

2.2.4 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 the intervals of a multiinterval.

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:

```
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 call returns all the neighboring 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 invariant. Due to the invariant, it won't have empty intervals, it will be sorted, and it won't have overlapping intervals.

3 GPT Algorithm

Kovács Attila and Forgács István proposed General Predicate Testing [2, p. 69]. It is a method of black-box test case generation based on and extending BVA.

On a high level, GPT works like BVA. We identify the boundaries and equivalence partitions of predicates. The difference is that the partitions are named (IN, ININ, ON, OFF, OUT) and we handle them as partitions, not just immediately select a test point from them.

Because we continue to handle them as intervals, after generating all partitions, we can reduce their number by identifying overlapping partitions and intersecting them.

Some terminology before we dive into GPT in detail:
- When we refer to intervals and intersections and equivalence partitions, they don't only mean numbers. Variables can have boolean or enum or any other custom types, but we can think about the values assignable to those as intervals. We can assign numbers to each value they represent and treat their values as single points.
- GPT only works with closed intervals (except for unbounded intervals, which are handled as unbounded). Because computers work with finite precision, we can convert an open interval to a closed interval. For example, if we use the precision of `0.01`, then the interval $[0, 1)$ can be converted to the closed interval $[0, 0.99]$.
- For simplicity, when we refer to intervals, we can either mean simple intervals or multiintervals. What matters is that they refer to value ranges and they can be intersected.

3.1 Boundary Value Analysis in GPT

3.1.1 Converting conditions to intervals

In GPT, we create intervals from predicates similar to Equivalence Partitioning. We create an interval, where all the elements inside the interval satisfy the predicate.

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

For booleans, we have a constant representation. If something is equal to some boolean, we take that boolean. If something is not equal to some boolean, we take that boolean and negate it. *Example:* For the predicate `x != true`, we will have `false`.

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

3.1.2 Extending BVA

Now that we have intervals to work with, we should generate the equivalence partitions and select the possible test values. In normal BVA, we do the partitioning and select single points from the intervals.

GPT extends this: We don't pick single values, but the largest possible intervals for the partitions. This will be helpful, when in the end we try to reduce the number of test cases, because we can see the overlap between different test cases. This ultimately helps us reduce the number of test cases required to test the same equivalence partitions.

3.1.3 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. In the examples the precision will be `0.01`.

- **IN:** The interval of the acceptable values. In case of unbounded 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:** The first possible acceptable values from the edges. These are single points, the endpoints of IN. Unbounded edges don't have ON points. If the interval is a single point, then 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:** The first not acceptable values from the edges. One step outside the edges of IN. Unbounded edges have no OF 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 with one step of precision, 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 OUT interval of $[10.02, \infty)$

Each of these can detect a different kind of predicate error. Because of one or two steps of precision, if there is an off-by-one error in the implementation (`>` instead of `>=` or the numbers are bigger or smaller) we can catch those.

Here the intervals of IN and ON overlap, why are we generating intervals which cover each other? Because in GPT, when we create the OFF and OUT intervals, we only do it for one variable at a time 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.

3.2 Test case generation in GPT

We can generate multiple intervals with GPT that we can select test values from. It has to be noted that this is just for one predicate. To test all the predicates in a condition, we use the following technique:

- Creating an NTuple from the condition.
- Generating the IN, ININ, and ON values for each variable in the NTuple.
- Generating the OFF and OUT values for each variable in the NTuple.

In the next sections I will show these steps in detail.

3.2.1 Creating an NTuple from the condition

An NTuple is a tuple with N elements. In GPT, I use the term NTuple for a map, where the keys are the variable names and the values are the EPs. This is because of historical reasons, originally these were literal tuples, but working with an explicit variable to EP mapping is easier to handle during graph reduction.

In GPT, we can only create NTuples from a condition that only contains conjunctions. I'll explain how to convert a condition with disjunctions to conjunctive forms in Section 4.5.

Example: The condition $x < 10$ && y in $[0, 20]$ && $z ==$ true would become the following NTuple:

```
{
   x: (-Inf, 10)
   y: [0, 20]
   z: true
}
```

3.2.2 Generating the IN, ININ, and ON values

Now we have an NTuple with variables that have EPs associated with them. We take the NTuple and for each variable we generate the IN, ININ, and ON values. Because the values are acceptable values, we can group them together and check all the variables' values in one NTuple. This won't be the case for OFF and OUT.

One complication is, that ON (and later OFF) can generate multiple values. When we could have multiple values for variables, we take the Cartesian product of the possibilities. This is further complicated by the fact that we could have N variables with M possible values, and we'd have to take the Cartesian product of all of those.

Example from the previous NTuple:

```
IN:   { x: (-Inf, 9.99], y: [0, 20], z: true }
ININ: { x: (-Inf, 9.98], y: [0.01, 19.99], z: true }
ON:   { x: 9.99, y: 0 and 20, z: true }
```

```
The Cartesian product of this is { x: 9.99, y: 0, z: true } and { x:
9.99, y: 20, z: true }
```

Example 2: Let's look at the ON and Cartesian product for the following NTuple:
`{ a: [0, 10], b: [20, 30] }`
The ON values would be: `{ a: 0 and 10, b: 20 and 30 }` The Cartesian products of this is:

```
{ a: 0, b: 20 }
{ a: 0, b: 30 }
{ a: 10, b: 20 }
{ a: 10, b: 30 }
```

3.2.3 Generating the OFF and OUT values

OFF and OUT values should not be acceptable by the SUT. To test that they are indeed not accepted, we'll generate OFF and OUT values one variable at a time. All the other variables will have IN values.

Example for the previous NTuple:

```
OFF:
   x:   { x: 10, y: [0, 20], z: true }
   y1: { x: (-Inf, 9.99], y: -0.01, z: true }
   y2: { x: (-Inf, 9.99], y: 20.01, z: true }
   z:   { x: (-Inf, 9.99], y: [0, 20], z: false }

OUT:
   x:   { x: [10.01, Inf), y: [0, 20], z: true }
   y1: { x: (-Inf, 9.99], y: (-Inf, -0.02], z: true }
   y2: { x: (-Inf, 9.99], y: [20.02, Inf), z: true }
   z:   { x: (-Inf, 9.99], y: [0, 20], z: false }
```

As you can see, OFF and OUT values can have multiple values. As I mentioned previously, we're taking the Cartesian product of these products. In reality, this means that we'll have two versions, because all the other IN values will become one value.

Note: In the algorithm, the resulting OFF and OUT NTuples aren't labelled with which variable they were generated for, I put it there in this example to make it easier to see.

3.3 Test value concretization in GPT

Test cases in GPT hold intervals of values for the variables. Programs need concrete values for variables to run test cases. To do this, we can just select either endpoint of the interval and that'll be a good test point. Test cases from GPT will always be closed intervals, so we can safely use those numbers. An exception is unbounded intervals, where we have to use the bounded part of that interval. If the interval is $(-\infty, \infty)$ we can choose 0.

The output values must be generated by the test designer manually. GPT only generates the inputs for the test cases. If it could also generate the outputs, that would mean that it could generate the correct implementation as well, which it can't.

4 GPT Lang

So far, we've looked at GPT as a test generation technique. The other feature of my GPT implementation is that I've developed a Domain Specific Language (DSL) for defining requirements for GP, I call it GPT Lang. From this DSL, GPT can generate test cases. This makes the test generation process much faster.

GPT Lang has a C inspired syntax, because this makes the language more intuitive for the developers. We can only write conditions in this DSL, since in GPT we are only concerned with predicates.

Let's look at an example:

```
var vip: bool
var price: num(0.01)
var second_hand_price: int

if(vip == true && price < 50) {
   if(second_hand_price == 2)
   if(second_hand_price in [6,8])
}

if(vip == false && price >= 50)
else if(vip == true || !(price >= 50 && second_hand_price >= 20)) {
   if(30 < price && 60 < second_hand_price)
}
else
```

As you can see, GPT Lang looks similar to how we will actually implement our programs. This can be useful in many more aspects. First, after we implement the requirements in GPT Lang and generate our test cases for Test Driven Development (TDD), we have a starting point for coding in other languages, as we have basically defined the control flow in GPT Lang. Also, if we have an existing codebase, we can easily test it with GPT, because we can convert the existing control flow to GPT Lang easily.

Currently, GPT Lang has the following functionality:

- Declare variables with boolean or number types (optionally with precision).
- Declare `if`, `else if`, and `else` statements, with optional bodies that can have other `if` statements. They can have any number of predicates.
- Declare predicates, which can be
  - Boolean `true` or `false`
  - Number $>, \geq, <, \leq, ==, \neq$ constant
  - Number `in` or `not in` interval
- Predicates can be negated with the prefix operator `!`, grouped with parentheses `( )`, conjoined with `&&` or disjointed with `||`.

4.1 Syntax

Numbers

Numbers can be either integers, floating point numbers, or `Inf` (for infinity). They can also be negative.

Example: `146`, `3.14`, `-6390`, `-Inf`, `0.01`

Type

Types can be:
- `bool`: Boolean.
- `int`: Integer.
- `num`: Number with default precision of 0.01.
- `num(<precision>)`: Number with the given precision. `<precision>` must be a positive number and not `Inf`.

Example: `bool`, `num`, `num(0.001)`, `num(1)`

Variable declaration

`var <var_name>: <type>`

Examples:
- `var price: num(0.01)`
- `var isVIP: bool`
- `var year: int`

Interval

`<lo_boundary> <lo>, <hi> <hi_boundary>`

Where
- `<lo_boundary>` is `(` or `[`
- `<lo>` and `<hi>` are numbers
- `<hi_boundary>` is `)` or `]`

Example: `(-Inf, 0]`, `[2.3, 6.75)`, `[1,1]`

Predicate

Predicates represent a comparison between variables and constants. They can either get evaluated to true or false.

Boolean predicate:
`<var_name> <op> <true|false>` or `<true|false> <op> <var_name>`
Where `<op>` is `==` or `!=`

Example: `isVIP == true`, `false != has_elements`

Binary number predicate:
`<var_name> <op> <constant>` or `<constant> <op> <var_name>`
Where `<op>` is `<`, `>`, `<=`, `>=`, `==`, or `!=`

Example: `price <= 10`, `0 < age`, `students != 0`

Interval predicate:
`<var_name> <in|not in> <interval>`

Example: `age not in [0, 18)`, `n in (-Inf, 200]`

Conditions

Conditions are predicates, joined together with `&&` or `||`, grouped with `( )` or a grouping is negated with `!( )`.

```
condition ::= <predicate>
            | (<predicate>)
            | !(<predicate>)
            | <condition> <&& | ||> <condition>
```

Examples:

- `price < 10 && age <= 18`
- `height > 170 || (age > 16 && parent_present == true)`
- `!(x == false && !(y == true || z == true))`

Ifs

If statements contain conditions they want to test, they work like most modern programming languages. They can have an optional body, which can have any number of if statements. They can have any number of `else if` branches, with their own conditions and body. Furthermore, they can have an optional `else` statements, which can also have a similar body.

```
if ::= if(<condition>) ({ <if>* })?
       (else if(<condition>) ({ <if>* })?)*
       (else ({ <if>* }))?
```

Examples:

- `if(x < 0) else`
- ```
  if(x > 0) {
      if(y == 0)
      else if(y == 1)
      else
  } else {
      if(y == -1)

      if(z == -1)
  }
  ```
  ```

4.2 Parsing and creating the AST

The AST of GPT Lang is the simple representation of the syntax. As it is a syntax tree, it has a tree structure with different nodes.

There are two main nodes: `VarNode` and `IfNode`.

`VarNode` holds a variable declaration. It has the variable name and it's type.

`IfNode` has it's own conditions as a `ConditionNode`, the body which contains a list of `IfNode`, a list of `ElseIfNodes` and an `ElseNode`.

`ConditionNode` is an ADT with three variants:
- A negated `ConditionNode`.
- An expression, which holds a single predicate.
- A group of expressions, with a left and right-hand side and an operand which is `||` or &&.

The parser is implemented using parser combinators. The output of the parser is this AST.

4.3 Converting the AST to IR

The AST is only a representation of the GPT Lang code. That structure can be brought to a simpler form, the Intermediary Representation (IR). We can also condense the condition structure, which will be the Reduced IR.

The IR has a similar structure for conditions. When we convert from the AST to the IR, it will only handle list of conditions. We have to flatten the `if` nodes, roll up the nested bodies, and handle the `else` and `else if` nodes.

These are the steps to convert an `IfNode`:
1. Convert the `if`'s conditions to IR conditions.
2. For the body, traverse recursively, take that result and conjunct the `if`'s conditions to it.
3. For the `else if` nodes, conjunct the top level conditions so far and conjunct the negated version to the `else if`s conditions.
4. For the body of `else if`s and the `else` statement, evaluate them recursively and conjoined the negated previous conditions onto them.

Example:

```
if(x == 1) {
   if(y == 2)
   if(y == 3)
} else if(x < 10)
else {
   if(z == 0)
}
```

In this case, from the body of the first `if`s we'll have:

```
x == 1
x == 1 && y == 2
x == 1 && y == 3
```

After that, we go to the `else if`. To get to that `else if`, the previous condition had to be false, so we conjoin that to the condition:

```
!(x == 1) && x < 10
```

To get to the `else` statement, both the `if` and the `else if` had to be false. We also append these to `if`'s body, so we'll have:

```
!(x == 1) && !(x < 10)
!(x == 1) && !(x < 10) && z == 0
```

In total, we'll have:

```
x == 1
x == 1 && y == 2
x == 1 && y == 3
!(x == 1) && x < 10
!(x == 1) && !(x < 10)
!(x == 1) && !(x < 10) && z == 0
```

Each of these lines is one Condition node inside the IR.

4.4 Flattening the IR

As you can see from the previous example, they could be reduced by a bit. This is where the Reduced IR comes in.

The IR is traversed recursively as a tree. When we have a negation node, we'll return the negated version of the condition node inside.

- The negation of negation nodes is the node itself.
- For negated expressions, we negate the contained expression. This is defined for intervals with the complement function. For booleans we swap `true` to `false` and `false` to `true`.
- For grouped expressions, we swap the operator (`||` to `&&` and `&&` to `||`) and we negate both the left-hand and right-hand sides. We evaluate them recursively. We can do this because of De Morgan's laws.

After this reduction, there will be no negation nodes in the IR.

Next, we'll flatten the expressions. Currently, they are stored in a binary tree. But a conjunction chain or disjunction chain can exist, where we have n elements, all of which are either all conjoined or disjointed. Our goal is to have a form, where if we have an `&&` node, it'll have any number of child nodes, but those child nodes are either single expressions or an `||` node. Same for the `||` node, it can only have simple expressions or `&&` nodes as children.

Example: Let's look at the following formula:

```
x > 0 && (x < 10 && y < 10 || x > 20 && y > 10 || z == 0) && y
== 2 && z > 20
```

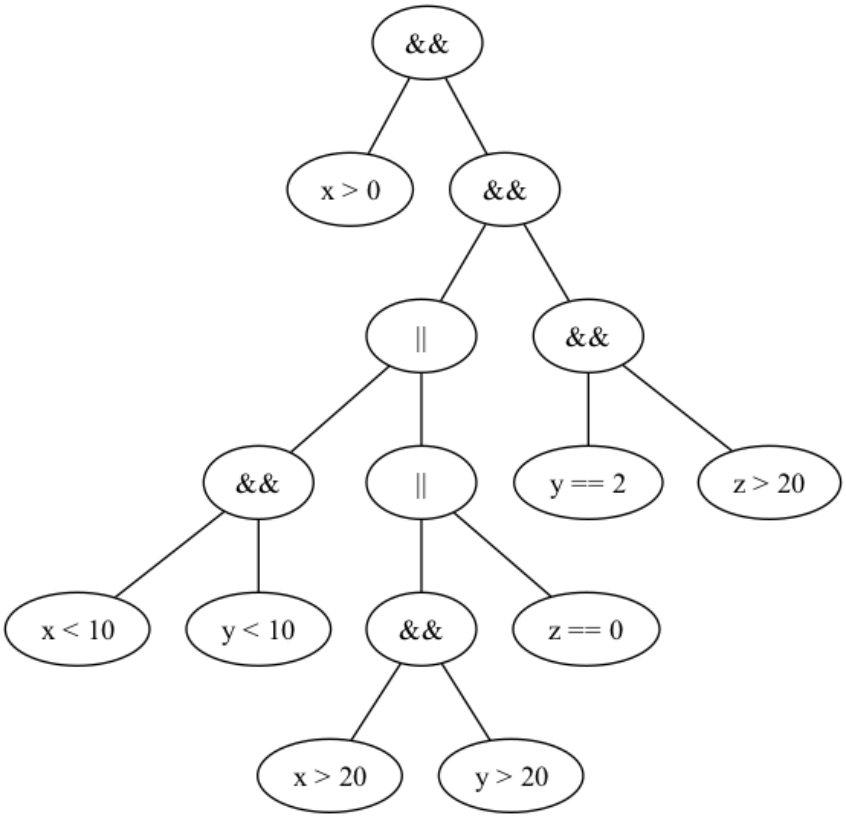

Figure 3: Before reduction

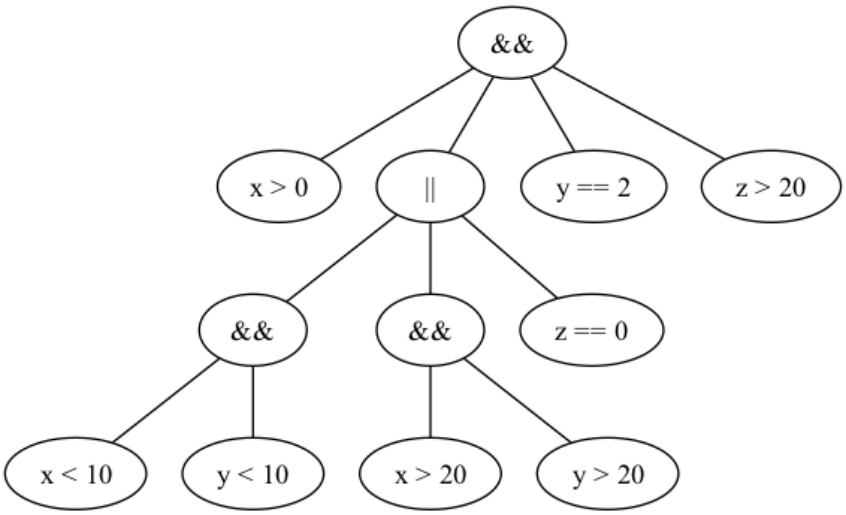

Figure 4: After reduction

As you can see, after the reduction we have an n-ary tree, where &&s only have simple predicates or || as children, and ||s only have simple predicates or &&s as children. For reference, I'll call this an AND-OR tree.

4.5 Converting disjunctions to conjunctions

Converting disjunctions to conjunctions is in my opinion one of the most important features I've come up with for GPT. The GPT algorithm is only defined for conditions where the predicates are conjoined. But in the real world, we rarely use only conjunctions, we have to use disjunctions as well.

There is a way to create conjunctions from disjunctions. If we think about it, when programs evaluate `x || y`, they always have an order of operations. Most programming languages today first evaluate `x`, if it is true, they stop and evaluate that condition as true. If `x` is false, they evaluate `y`. We can use this order of operation to define two conjunctions: `x` and `!x && y`. If we write test cases for these two conjunctions, it is the same as if we tested `x || y`. Now we can use GPT on these converted predicates.

We need to make a slight adjustment for this. Because we are black box testing, we can't assume anything about the implementation. This includes the order in which the `||` is evaluated in or how the programmer implements it. Implementing `x || y` should have the same meaning as `y || x`, if we are testing idempotent pure functions that have no side effects.

Example: If we only generate test cases for `x` and `!x && y` and the programmer implements `y || x`, we don't test the case when only `y` is true or `!y && x`.

Based on this, we can say that we want to take the linear combination of the conditions inside a disjunction and generate all possible combinations.

Example: Let's look at the disjunctions generated for `x || y || z`:

```
x
!x && y
!x && !y && z
!x && z
!x && !z && y
y
!y && x
!y && z
!y && !z && x
z
!z && x
!z && y
```

As we can see, we didn't generate `!z && !x && y`, because we've already generated `!x && !z && y` and these would semantically be the same, since the order of elements can be switched around in conjunctions. This is also why z has fewer cases, because they've been generated previously.

This method is applied recursively, because we could have && conditions inside ||s. Also, when we have an && condition with an || sub-condition, the linear combinations are generated 'in their place', the 'outer context' will remain the same.

Example: `x && (y || z) && w`

```
x && y && w
x && !y && z && w
x && z && w
x && !z && y && w
```

In this case, the 'outer context' had one element, always `x && w`, it didn't have any variations. But what if it consisted of multiple variations, because they were `||`s too? We have to take the Cartesian product of those variations.

Example: `(a || b) && (c || d)`

```
a && c
a && !c && d
a && d
a && !d && c
!a && b && c
!a && b && !c && d
!a && b && d
!a && b && !d && c
b && c
b && !c && d
b && d
b && !d && c
!b && a && c
!b && a && !c && d
!b && a && d
!b && a && !d && c
```

We can see, that this is the Cartesian product is the same, as if we had generated `a || b` and `c || d` separately.

Because of the recursive nature of these methods, they can be applied to any deep and wide AND-OR tree.

The downside of this method is that it generates a huge amount of conditions and in turn a huge amount of test cases. But this is the price we pay for black box testing. This way we can guarantee that the implementation can have the predicates inside the `||`s in any order, we'll have a test case for it. Since our end goal is to make sure that we that the program adheres to the requirements, this is the correct approach.

This code could be modified, so that it doesn't generate all the variable orders for `||`s. The test designer and programmer could agree on the order of the variables and that'd reduce the number of test cases if needed. Later, it could cause regressions if the variable order is changed, so this is not recommended.

5 Graph Reduction

5.1 What is Graph Reduction?

When GPT generates test cases, it generates an interval (or boolean value) for variables. This means, that the discrete test points should come from those intervals. But there could be a case when multiple test cases would have intersecting intervals. For example:

- T1: `{x: [0, 10]}`
- T2: `{x: (-10, 5)}`
- T3: `{x: [0, 200]}`

For example, in this case, if we select $x = 2$ as out test point, it would cover all the three test cases.

This way, we can reduce the number of our total test cases by trying to find test cases (NTuples) that intersect, then calculate their intersection.

In this example, the intersection of T1 and T2 is `{x: [0, 5)}` and the intersection of that and T3 is `{x: [0, 5)}`.

As we can see, these intersections are still correct and valid test cases, we are still covering all the original test cases generated by GPT. Graph reduction won't violate the correctness of the test cases.

5.1.1 NTuple intersection

First, I'll have to clarify that booleans only intersect when their values are the same. So, `true` intersects with `true`, and `false` with `false`. This can be derived from intervals, for example, if `true` is $[0, 0]$ and `false` is $[1, 1]$. This method can also be applied to Enums (which is a future improvement idea, as detailed in Section 9.4).

Two NTuples intersect, when all of their variables intersect. If a variable is not present in an NTuple, we treat it as if it could take all the possible values. In practice this means, that we just use the value of that variable from the other NTuple.

Example: The intersection of `{x: [0, 100], y: false, z: [5,5]}` and `{z: [10, 20], y: false}` is `{x: [0, 100], y: false, z: [5,5]}`.

Because NTuples have intersections, in the following sections I'll give examples only with simple intervals, to keep them concise. But those examples can be used for NTuples as well.

5.1.2 Graph representation

Our goal is to intersect NTuples in a way, that in the end we have the least number of NTuples in the end. We can imagine this problem as a Graph, where the nodes are the NTuples, and we have an edge between two nodes, if those NTuples have an intersection.

Example:

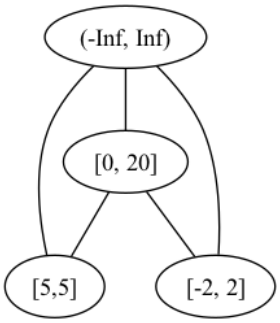

Figure 5:

What happens when we intersect two nodes?

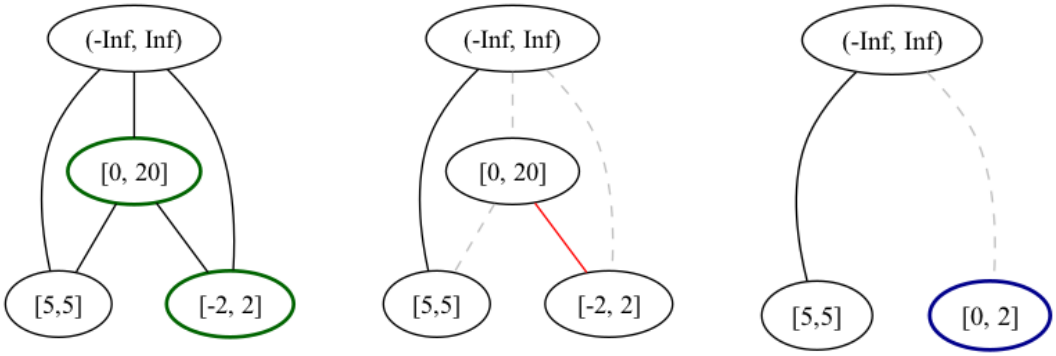

Figure 6: One intersection in graph reduction

In Figure 6 we can see, that we select $[0, 20]$ and $[-2, 2]$ for intersection. The detailed process for intersection is the following:

1. We identify the edge connecting them. *(red)*
2. We identify the edges and nodes they are connected to. *(gray dashed)*
3. We remove the edges connecting them to other edges *(gray dashed)* and remove the edge connecting them. *(red)*
4. We intersect the nodes, and replace the two original nodes with the new intersected node. *(blue)*
5. We look at the nodes they were originally connected to *(gray dashed)* and see if the new intersected node intersects them. If yes, we restore those edges.

In step 5, it is enough to look at the edges, which were originally connected to the nodes, instead of the whole graph. This is, because with intersections our intervals can only get smaller, so we'd never add new edges to the graph that weren't there before.

5.1.3 Strategies for optimizing Graph Reduction

This graph reduction is an NP-complete problem [2, p. 116]. Because of this, we can only create algorithms that approximate the most optimally reduced graph.

As you can see, every step of our graph reduction basically creates a new graph. We remove and replace nodes, and also we remove edges. This poses a problem, because the order of reduction matters. Consider the following example:

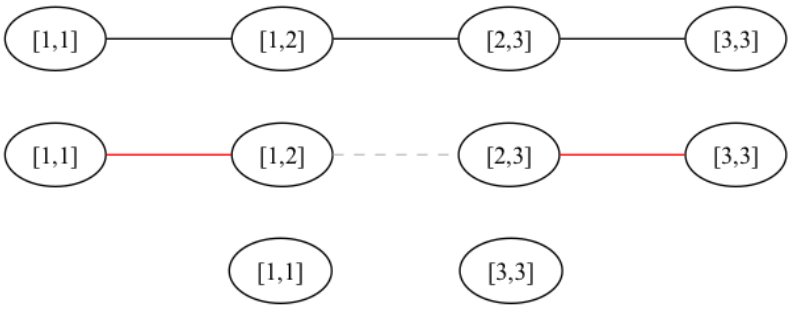

Figure 7: Optimal join

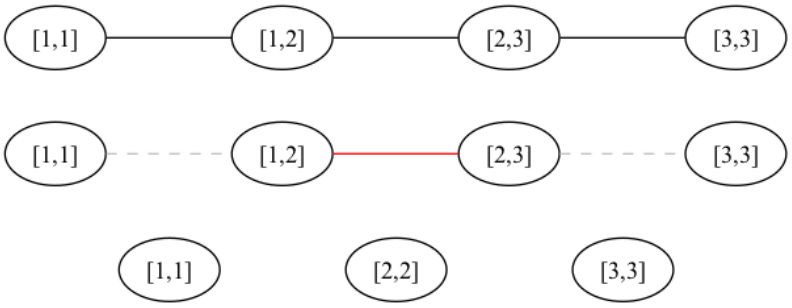

Figure 8: Not optimal join

If this problem was one dimensional, as in, we only had simple intervals, we could use the $O(n \log n)$ solution proposed in [26]. However, this algorithm assumes that the intervals can be sorted in increasing order by their hi endpoints. Because we are dealing with NTuples, and we could have n intervals for each interval, we can't use this algorithm, because we can't define a partial ordering on it that'd work in this case.

5.1.4 Abstracting Graph Reduction

Graph reduction can be posed in a more abstract and generic way. This allows us to concentrate more on the reduction part, without getting lost in the details.

After we construct our graph (with the intersection of NTuples), we don't actually need to know the contents of the nodes. In the previous section, the definition for joining edges doesn't actually need to know whether the nodes intersect or not, they only need to know how the edges are connected. (That was derived from intersections, but we can focus on generic edges and nodes from now on.)

We can rephrase the problem the following way: Given a graph, join nodes on edges until no edges remain in the graph. When joining two nodes, the joint node will have edges to nodes to which both the original nodes had edges to. If there were

nodes that only one of the original nodes had edges to, then the joint node won't have that edge.

Terminology:
- Losing an edge: When joining nodes, the joint node won't have an edge to a node to which the original nodes had edges to.
- Retaining nodes: When joining nodes, the joint node will have an edge to a node to which the original nodes had edges to.

With this, we can define some properties.

1. When joining nodes, the joint node will retain edges to nodes, to which both the joint nodes have edges to.
2. When joining nodes, edges will be lost to nodes, to which only one joining node had edges to.
3. If we have N nodes which all have edges between them, so it is a complete graph, they can be reduced down to one node. This comes from point 1. and 2., because we only lose edges where not all nodes have edges to that one node, but because we have a complete graph we don't lose edges.
4. We can freely reduce nodes where we retain all the edges after the join.

Hypothesis: There is an optimal way to reduce an acyclic component of nodes. The optimal way is, to start joining nodes "from the edges", meaning, from nodes which only have one edge. We can trace back all acyclic components to the example with 4 nodes joined in a line in Figure 7 and Figure 8.

In Figure 7, we're first joining edges from the ends. This means, that we only lose one edge when joining. In Figure 8, when we are joining in the middle, we lose two edges. Due to this, my hypothesis is that joining from the edges is a good strategy.

As we can see, we can reduce the same starting point to either 2 or 3 three nodes in the end. This means, that there will always be an optimal solution to this problem, and suboptimal ones.

Components with more than 4 nodes exhibit the same behavior. There are also cases, where a component can split into two 4 node chains (like in the example) and then the difference between the optimal and not optimal solutions is 2 (one for each 4 node segment). This can be achieved with a 10 node chain and joining the two nodes in the middle.

5.2 MONKE

MONKE stands for Minimal Overhead Node Kollision (Collision) Eliminator. This means, that this algorithm has little to no heuristics (no overhead) and it eliminates colliding (intersecting) nodes, i.e. joining them.

MONKE works in a very simple way: We take the list of edges in the graph, always take one and join the nodes on the edge. Repeat, until there are no edges in the graph.

Surprisingly, this algorithm works quite well, as we'll see in Section 5.6.

Hypothesis: If there was a seriously wrong way to join nodes, as in, lose a lot of nodes that other joins could retain, then MONKE wouldn't work that well. Because MONKE is essentially a random algorithm, it wouldn't give optimal results. Therefore, my hypothesis is, that there are no completely wrong joins that can be made. Yes, as we've seen in the previous section, there are ways to have an optimal reduction, but suboptimal reductions are rare.

5.3 Least Losing Nodes Reachable

The heuristic of the Least Losing Nodes Reachable (LLNR) algorithm is that it tries to lose the least amount of nodes that could be reached with a Breadth First Search (BFS).

The idea behind this algorithm is, that if we lose the least number of nodes reachable, then we can join the nodes which can be 'freely joined' (as described in the previous chapter).

The algorithm assigns a weight to each edge, with the following steps:

1. Start a BFS from one of the edges of the node. Count how many nodes are reachable. This will be the before count.
2. Copy the graph and join the nodes on that edge.
3. Do a BFS in the cloned graph from the joint node. The node count will be the after count.
4. `before - after` will be the number of nodes that we wouldn't be able to reach after the join. This will be weight of the edge.

We calculate the weights of all edges. We join the edge with the smallest weight (the one that loses the least nodes reachable). After a join, all the edge weights have to be recalculated, because we potentially lost edges, so graph traversal is effected.

A further optimization could be, that we only recalculate the edges inside a component, because other components' graph traversal won't be affected.

This algorithm is much more computationally expensive than MONKE. Before each join, we have to recalculate all edge weights, so for each edge we have to do a BFS. This scales exponentially with the number of edges in the graph.

Another downside of the algorithm is that it only considers one join. It can't see that a join might be disadvantageous a number of steps from now.

5.4 Least Losing Edges

The heuristic for Least Losing Edges (LLE) is that after each join, we want to lose the least number of edges in total in the graph.

The idea behind this algorithm is similar to Least Losing Nodes Reachable: we want to make optimal joins first.

The algorithm works similarly as Least Losing Nodes Reachable. It assigns a weight to each edge with the following steps:

1. Count the number of edges in the graph.
2. Copy the graph and join the nodes on the edge. Count how many edges are in the copied graph.
3. Count how many edges were removed (lost) from the graph due to the join.

We calculate the weights of all the edges. We join the edge with the smallest weight (the one that loses the least edges). After a join, all the edge weights have to be recalculated, because subsequent joins could affect the edges differently, than the number we calculated.

This algorithm is also much more computationally expensive than MONKE, but a bit faster than Least Losing Nodes Reachable. For each join, we have to simulate all the joins (Same as LLNR), but we only have to count the number of edges, which is a less expensive operation than doing a BFS twice. The number of nodes can be stored in a variable and modified after each join, so it could be a constant operation. This algorithm scales exponentially with the number of edges.

It has the same downside as LLNR, because it only considers the one join, not the subsequent steps.

5.4.1 Most Losing Edges

A variation of the LLE algorithm is the Most Losing Edges (MLE) algorithm. It works the exact same way as LLE, but when selecting the edge to join, it'll select the one with the highest weight.

This algorithm is absolutely not practical, but is a good frame of reference for the other algorithms. We can approximate how good a reduction is by looking at a "worst case" reduction.

Hypothesis: MLE approximates the worst case of graph reduction. It'll always try to join edges which'll lose the most edges, which are probably bad joins.

5.5 Least Losing Components

The heuristic for Least Losing Components (LLC) is that after each join we don't want the graph to split into a lot of components. If our graph splits into components, then we can't join nodes in different components. If the graph stays in the fewest number of components, the potential for joins could be there.

The idea behind this algorithm is similar to Least Losing Nodes Reachable, we want to make optimal joins first.

The algorithm works similarly as Least Losing Nodes Reachable. It assigns a weight to each edge with the following steps:

1. Count the number of components in the graph.
2. Copy the graph and join the nodes on the edge. Count how many components are in the copied graph.
3. Count how many new components would be in the graph due to the join.

We calculate the weights of all the edges, then we join the edge with the smallest weight (the one which loses the components). After a join, all the edge weights have to be recalculated, because subsequent joins could affect the edges differently, than the number we calculated.

This algorithm is very computationally expensive. For each join, we have to simulate all the joins (Same as LLNR), but we only have to count the number of components in the graph, which is a costly operation.

It has the same downside as LLNR, because it only considers the one join, not the subsequent steps.

5.6 Comparing the Graph Reduction Algorithms

For the benchmark, I've used a 2020 Apple M1 MacBook Pro with 16Gb or RAM. I've used cargo-instrument [27] that uses XCode Instruments [28] for profiling.

The baseline runs represent the baseline, when no graph reduction is done. It is needed, because I measured the whole runtime of the program, which includes parsing the GPT Lang source file, creating the NTuples, and creating the initial graph. To get the runtime of only the algorithms, you can subtract the baseline runtime from the runtime.

How to read the results:

- The **Runtime** show the total runtime of the program in seconds (s) or milliseconds (ms). Lower is better.
- The **Num. of Test Cases** columns shows the reduced number of test cases. Lower is better.
- The **%** columns show the reduction percentage. Higher is better.
- The table is ordered by runtime in ascending order.
- When comparing percentages I use percentage points (pp for short).

5.6.1 Benchmarks for price_calculation.gpt

GPT Lang code can be found in Appendix A.2

Number of non-reduced test cases: 14
Number of edges in the initial graph: 39

Algorithm	Runtime	Num. of Test Cases	%
baseline	1ms	14	0%
MONKE	3ms	8	42.86%
MLE	3ms	9	35.71%
LLE	3ms	8	42.86%
LLNR	3ms	8	42.86%
LLC	4ms	8	42.86%

Table 1: price_calculation.gpt benchmark results

Price Calculation is a really simple example, with a tiny graph of 39 edges. All the algorithms reduce to the same number of test cases, except for MLE, as expected.

The runtime is also similar, because of the small graph.

5.6.2 Benchmarks for paid_vacation_days.gpt

GPT Lang code can be found in Appendix A.1

Number of non-reduced test cases: 55
Number of edges in the initial graph: 183

Algorithm	Runtime	Num. of Test Cases	%
baseline	3ms	55	0%
MONKE	3ms	22	60%
MLE	5ms	29	47.27%
LLE	9ms	22	60%
LLC	49ms	24	56.36%
LLNR	57ms	22	60%

Table 2: paid_vacation_days.gpt benchmark results

This graph is almost 4.7 times the Price Calculation graph. Now, we can start to see a difference between the number of test cases. MONKE, LLE, and LLNR produce the best reduction. LLC can't reduce that well with 3.64 pp behind the best ones. MLE performs the worst, as expected. However, it is worth noting, that MLE could still reduce the graph almost by half and is 12.73 pp behind the best.

We can see the exponential factor coming in for LLC and LLNR. Their runtimes are an order of magnitude worse than the other algorithms.

5.6.3 Benchmarks for complex_small.gpt

GPT Lang code can be found in Appendix A.3

Number of non-reduced test cases: 328
Number of edges in the initial graph: 11684

Algorithm	Runtime	Num. of Test Cases	%
baseline	0.028s	328	0%
MONKE	0.031s	51	84.45%
MLE	3.11s	71	78.35%
LLE	26.84s	45	86.28%
LLC	130.80s	54	83.54%
LLNR	216.60s	53	83.84%

Table 3: complex_small.gpt benchmark results

Here we make the jump from 189 edges to 11 684 edges and the difference is striking. The runtime of MONKE is still around the baseline with 31ms. We can see that the other algorithms are slower: MLE is 100 times, LLE is 865 times, LLC is 4219 times and LLNR is 6967 slower than the baseline. The exponential scaling difference can be clearly seen in these results.

The difference between the reductions may seem significant at first. LLE has the best reduction, and while MONKE may seem to have 10% more test cases than LLE, the overall reduction from the starting 328 test cases is just 1.83 pp. With MONKE being almost 3 orders of magnitude faster, this 1.83 pp difference is not that much in comparison.

It is also interesting, that MLE has 1.5 times as many test cases as LLE, but the overall difference between reduction is 7.93 pp.

5.6.4 Benchmarks for complex_medium.gpt

GPT Lang code can be found in Appendix A.4

Number of non-reduced test cases: 452
Number of edges in the initial graph: 23664

Algorithm	Runtime	Num. of Test Cases	%
baseline	0.066s	452	0%
MONKE	0.084s	69	84.73%
MLE	11.19s	99	78.1%
LLE	111.00s	58	87.17%

Table 4: complex_medium.gpt benchmark results

Because of the exponential scaling of LLC and LLNR, I wasn't able to run them, because they'd take too much time.

In this benchmark, the number of edges in the graph is doubled. The relative difference between the results is similar to the previous example. LLE is 1321 times slower than MONKE. MONKE also starts to drift from the baseline, which means that scaling starts to kick in as well.

There is a 2.44 pp difference between MONKE and LLE.

5.6.5 Benchmarks for complex_hard.gpt

GPT Lang code can be found in Appendix A.5

Number of non-reduced test cases: 3657
Number of edges in the initial graph: 1229629

Algorithm	Runtime	Num. of Test Cases	%
baseline	47.37s	3657	0%
MONKE	153.00s	354	90.32%

Table 5: complex_hard.gpt benchmark results

Because of the exponential scaling of the other algorithms, I could only run MONKE in a reasonable amount of time.

In this example the 1.2 million edges in the graph is quite a big jump from the 22 thousand previously. It shows on MONKE's runtime as well.

From the profiling information, I saw that 108 seconds were spent removing the nodes from the graph. If we subtract the baseline's from MONKE's, that means that most of the runtime is spent removing nodes from the graph. A future improvement idea would be to find a Graph structure that has low costed node removal, while also keeping the node joining performance, so neighbor lookup and edge addition is fast.

5.6.6 Summary

From these benchmarks we could see, that MONKE is the most performant and the second best performing reduction algorithm. In reduction per seconds of runtime it is the best algorithm.

MLE achieving 78% reduction also supports my hypothesis, that there is likely no absolutely wrong way to join nodes that would affect the end result of the reduction significantly. There are certainly better ways and heuristics to use when reducing, as we can see with LLE. The test designers should decide how much time they are willing to spend waiting for test case generation, with the benefit of having a more reduced test set.

In the last example the number of test cases have been reduced to a tenth of the baseline. This shows the power and usefulness of graph reduction. There is quite a difference between having to run 3657 versus 354 test cases, when they cover the exact same equivalence partitions.

6 Validation

In this chapter, I'll validate that my automatic test generation algorithm works correctly. I'll compare my generated test cases to the ones shown in the book [2], so we can see if my algorithm differs from the proposed manual GPT method by Kovács Attila and Forgács István.

6.1 Comparing generated test cases for Paid Vacation Days

We'll look at the Paid Vacation Days example [2, p. 100].

Paid vacation days

- R1: The number of paid vacation days depends on age and years of service. Every employee receives at least 22 days per year.

Additional days are provided according to the following criteria:

- R2-1: Employees younger than 18 or at least 60 years, or employees with at least 30 years of service will receive 5 extra days.
- R2-2: Employees of age 60 or more with at least 30 years of service receive 3 extra days, on top of possible additional days already given based on R2-1.
- R2-3: If an employee has at least 15 years of service, 2 extra days are given. These two days are also provided for employees of age 45 or more. These extra days cannot be combined with the other extra days.

Display the vacation days. The ages are integers and calculated with respect to the last day of December of the current year.

Let's look at the if statements one by one and compare the generated test cases.
Note: this is without graph reduction. The test cases from the book will have an ID starting with B, my GPT implementation will have an M prefix.

The full GPT Lang definition can be found in Appendix A.1.

R1-1: IF age < 18 AND service < 30 THEN...

In the book, the test cases are:

No.	Test Case
B1	ON: age = 17, service = 29
B2	OFF1: age = 18, service < 30
B3	IN: age << 18, service << 30
B4	OUT1 age > 18, service < 30
B5	OFF2: age < 18, service = 30
B6	OUT2: age < 18, service > 30

Table 6: R1-1 (1) from the book

With my GPT:

No.	Test Case	Book No.
M1	age: $(-\infty, 17]$, service: $(-\infty, 29]$	-
M2	age: $[17, 17]$, service: $[29, 29]$	B1
M3	age: $(-\infty, 16]$, service: $(-\infty, 28]$	B3
M4	age: $(-\infty, 17]$, service: $[30, 30]$	B5
M5	age: $(-\infty, 17]$, service: $[31, \infty)$	B6
M6	age: $[18, 18]$, service: $(-\infty, 29]$	B2
M7	age: $[19, \infty)$, service: $(-\infty, 29]$	B4

Table 7: R1-1 (1) with my GPT

As we can see, my GPT covers all the test cases in the book, but has an additional test case: M1. This is there, because in the book B3 is said to be an IN, but it is actually an ININ. M1 in this case is the IN. This is, because in the book IN and ININ are not differentiated this concretely. Because ININ is a subset of IN, it is actually enough to generate the ININ for an interval.

I'm generating both the IN and the ININ, because there could be intervals which have an IN but not ININ (for example $[0, 0.1]$ if the precision is 0.1.). It is easier to generate both the IN and ININ and let graph reduction take care of joining the intervals.

R1-1: IF age ≥ 60 AND service < 30 THEN...

In the book, the test cases are:

No.	Test Case
B7	ON: age = 60, service = 29
B8	OFF1: age = 59, service < 30
B9	OFF2: age ≥ 60, service = 30
B10	IN: age > 60, service << 30
B11	OUT1 age << 60, service < 30
B12	OUT2: age ≥ 60, service > 30

Table 8: R1-1 (2) from the book

With my GPT:

No.	Test Case	Book No.
M8	age: $[60, \infty)$, service: $(-\infty, 29]$	-
M9	age: $[60, 60]$, service: $[29, 29]$	B7
M10	age: $[61, \infty)$, service: $(-\infty, 28]$	B10
M11	age: $[59, 59]$, service: $(-\infty, 29]$	B8
M12	age: $(-\infty, 58]$, service: $(-\infty, 29]$	B11
M13	age: $[60, \infty)$, service: $[30, 30]$	B9
M14	age: $[60, \infty)$, service: $[31, \infty)$	B12

Table 9: R1-1 (2) with my GPT

As we can see here as well, my GPT covered all the test cases from the book. The additional test case M8 is the IN, for the same reason as previously.

R1-1: IF service ≥ 30 AND age < 60 AND age ≥ 18 THEN...

In the book, the test cases are:

No.	Test Case
B13	OUT1: age << 18, service ≥ 30
B14	OFF1: age = 17, service ≥ 30
B15	ON1/IN2: age = 18, service > 30
B16	ON2: age = 59, service = 30
B17	OFF2: age < 60 && age ≥ 18, service = 29
B18	OUT2: age < 60 && age ≥ 18, service << 30
B19	OFF3: age = 60, service ≥ 30
B20	OUT3: age > 60, service ≥ 30

Table 10: R1-1 (3) from the book

With my GPT:

No.	Test Case	Book No.
M15	age: $[18, 59]$, service: $[30, \infty)$	-
M16	age: $[18, 18]$, service: $[30, 30]$	-
M17	age: $[59, 59]$, service: $[30, 30]$	B16
M18	age: $[19, 58]$, service: $[31, \infty)$	-
M19	age: $[18, 59]$, service: $[29, 29]$	B17
M20	age: $[18, 59]$, service: $(-\infty, 28]$	B18
M21	age: $[17, 17]$, service: $[30, \infty)$	B14
M22	age: $[60, 60]$, service: $[30, \infty)$	B19
M23	age: $(-\infty, 16]$, service: $[30, \infty)$	B13
M24	age: $[61, \infty)$, service: $[30, \infty)$	B20

Table 11: R1-1 (3) with my GPT

Here we can see a difference between my GPT and the book. In my implementation, the two different predicates for age (age < 60 AND age ≥ 18) get merged to one interval: $[18, 60)$.

For B15, I don't have an exact test case. This is because B15 combined the ON1 and the IN2. I have a separate test case for ON1 with M16 and a separate one for IN2 with M18.

M15 is the IN, as discussed previously.

All in all, with analyzing B15, we can say that my test cases cover the ones in the book.

R1-2: IF service ≥ 30 AND age ≥ 60 THEN...

In the book, the test cases are:

No.	Test Case
B21	OUT1: age << 60, service ≥ 30
B22	OFF1: age = 59, service ≥ 30
B23	ON: age = 60, service = 30
B24	OFF2: age ≥ 60, service = 29
B25	IN: age > 60, service > 30
B26	OUT2: age ≥ 60, service << 30

Table 12: R1-2 from the book

With my GPT:

No.	Test Case	Book No.
M25	age: $[60, \infty)$, service: $[30, \infty)$	-
M26	age: $[60, 60]$, service: $[30, 30]$	B23
M27	age: $[61, \infty)$, service: $[31, \infty)$	B25
M28	age: $[60, \infty)$, service: $[29, 29]$	B24
M29	age: $[60, \infty)$, service: $(-\infty, 28]$	B26
M30	age: $[59, 59]$, service: $[30, \infty)$	B22
M31	age: $(-\infty, 58]$, service: $[30, \infty)$	B21

Table 13: R1-2 with my GPT

All the test cases from the book are covered. M25 is the IN, as mentioned previously.

R1-3: IF service ≥ 15 AND age < 45 AND age ≥ 18 AND service < 30 THEN...

In the book, the test cases are:

No.	Test Case
B27	OUT1: age << 18, service ≥ 15 && service < 30
B28	OFF1: age = 17, service ≥ 15 && service < 30
B29	OFF2: age < 45 && age ≥ 18, service = 14
B30	ON1: age = 18, service = 15
B31	OFF3: age < 45 && age ≥ 18, service = 30
B32	OUT2: age < 45 && age ≥ 18, service > 30
B33	OUT3: age < 45 && age >= 18, service << 15
B34	ON2: age = 44, service = 29
B35	OFF4: age = 45, service ≥ 15 && service < 30
B36	OUT4: age > 45, service ≥ 15 and service < 30

Table 14: R1-3 (1) from the book

With my GPT:

No.	Test Case	Book No.
M32	age: $[18, 44]$, service: $[15, 29]$	-
M33	age: $[18, 18]$, service: $[15, 15]$	B30
M34	age: $[44, 44]$, service: $[15, 15]$	-
M35	age: $[18, 18]$, service: $[29, 29]$	-
M36	age: $[44, 44]$, service: $[29, 29]$	B34
M37	age: $[19, 43]$, service: $[16, 28]$	-
M38	age: $[18, 44]$, service: $[14, 14]$	B29
M39	age: $[18, 44]$, service: $[30, 30]$	B31
M40	age: $[18, 44]$, service: $(-\infty, 13]$	B33
M41	age: $[18, 44]$, service: $[31, \infty)$	B32
M42	age: $[17, 17]$, service: $[15, 29]$	B28
M43	age: $[45, 45]$, service: $[15, 29]$	B35
M44	age: $(-\infty, 16]$, service: $[15, 29]$	B27
M45	age: $[46, \infty)$, service: $[15, 29]$	B36

Table 15: R1-3 (1) with my GPT

All the test cases from the book are covered by my GPT.

M32 is the IN as explained previously.

M33 and M35 are ON points. My GPT merges the predicates for service and treats it as $[15, 30)$ and age as $[18, 45)$. When generating ON points, age will have 18 and 44, service will have 15 and 29. My GPT takes the Cartesian product of these, that's why M34 and M35 appeared, in addition to M36, which is in the book as B34.

M37 is the ININ of the intervals. The book says that the ON points are also IN points, that's why there are no explicit IN intervals. As discussed previously, my GPT generates both IN and ININ points, so this ININ appeared, which is not a problem, as it is a valid test case.

R1-3: IF age ≥ 45 AND service < 30 AND age < 60 THEN...

In the book, the test cases are:

No.	Test Case
B37	OUT1: age << 45, service < 30
B38	OFF2: age = 44, service < 30
B39	ON1: age = 45, service = 29
B40	OFF2: age ≥ 45 && age < 60, service = 30
B41	OUT2: age ≥ 45 && age < 60, service > 30
B42	ON2: age = 59, service << 30
B43	OFF3: age = 60, service < 30
B44	OUT: age > 60, service < 30

Table 16: R1-3 (2) from the book

With my GPT:

No.	Test Case	Book No.
M46	age: $[45, 59]$, service: $(-\infty, 29]$	-
M47	age: $[45, 45]$, service: $[29, 29]$	B39
M48	age: $[59, 59]$, service: $[29, 29]$	-
M49	age: $[46, 58]$, service: $(-\infty, 28]$	-
M50	age: $[44, 44]$, service: $(-\infty, 29]$	B38
M51	age: $[60, 60]$, service: $(-\infty, 29]$	B43
M52	age: $(-\infty, 43]$, service: $(-\infty, 29]$	B37
M53	age: $[61, \infty)$, service: $(-\infty, 29]$	B44
M54	age: $[45, 59]$, service: $[30, 30]$	B40
M55	age: $[45, 59]$, service: $[31, \infty)$	B41

Table 17: R1-3 (2) with my GPT

All the test cases in the book are covered, except for B42. B42 is the combination of an ON and an ININ. The ININ for service (and age) is M49. The ON for age is M48.

M46 is the IN, same as previously.

Summary

In total, the book has 44 test cases, my GPT generated 55 test cases.

The 11 test cases not in the book are:
- M1, M8, M15, M25, M32, M46 are INs (+6 test cases)
- M37 is an ININ (+1 test case)
- B42 -> M47 + M48 (+1 test case)
- B15 -> M16 + M18 (+1 test case)
- M34 and M35 are additional ONs (+2 test cases)

After graph reduction

With my GPT and Least Losing Edges:

No.	Test Case
#1	age: $[17, 17]$, service: $[29, 29]$
#2	age: $[45, 45]$, service: $[29, 29]$
#3	age: $(-\infty, 16]$, service: $[15, 28]$
#4	age: $[18, 18]$, service: $[29, 29]$
#5	age: $[46, 58]$, service: $[15, 28]$
#6	age: $[17, 17]$, service: $[30, 30]$
#7	age: $(-\infty, 16]$, service: $[31, \infty)$
#8	age: $[44, 44]$, service: $[15, 15]$
#9	age: $[44, 44]$, service: $[29, 29]$
#10	age: $[18, 18]$, service: $[15, 15]$
#11	age: $[59, 59]$, service: $[29, 29]$
#12	age: $[18, 44]$, service: $[31, \infty)$
#13	age: $[18, 44]$, service: $[14, 14]$
#14	age: $[61, \infty)$, service: $[31, \infty)$
#15	age: $[18, 44]$, service: $(-\infty, 13]$
#16	age: $[18, 18]$, service: $[30, 30]$
#17	age: $[59, 59]$, service: $[30, 30]$
#18	age: $[45, 58]$, service: $[31, \infty)$
#19	age: $[60, 60]$, service: $[29, 29]$
#20	age: $[60, 60]$, service: $[30, 30]$
#21	age: $[61, \infty)$, service: $(-\infty, 28]$
#22	age: $[19, 43]$, service: $[16, 28]$

Table 18: Final reduced test cases for Paid vacation Days

Conclusion

In the book, the number of reduced test cases is 18. This is 40.9% of the original 44 test set.

My GPT with MONKE reduced the number of test cases to 22. This is 40% of the original test set.

In conclusion, my GPT implementation generated all the test cases that were mentioned in the book. It also generated a few more test cases, but most of them can be reduced with graph reduction. The graph reduction reduced the number of test cases by a similar percentage than the reduction in the book.

Hypothesis: The 4 additional test cases in the reduced test set are due to M34, M35 (ONs), the breaking of B42 into M47 and M48, and breaking B15 into M16 and M18. These are additional test cases which weren't in the book. These NTuples (test cases) can't be intersected with other NTuples, so they remain in the reduced graph. We can see, that #8 is M34, #4 is M35. I couldn't trace back the effect of B42 and B15.

6.2 Catching predicate errors in Paid Vacation Days

In the book Paid Vacation Days has the following implementation [2, p. 106], I rewrote it in Javascript.

```
function paidVacationDays(age, service) {
   let days = 22 /* 1 */

   if (age < 18 /* 2 */ || age >= 60 /* 3 */|| service >= 30 /* 4
*/) {
      days += 5 /* 5 */
   }
   if(age >= 45 /* 6 */ && age < 60 /* 7 */ && service < 30 /* 8
*/) {
      days += 2 /* 9 */
   }
   if(age >= 18 /* 10 */ && age < 45 /* 11 */ && service >= 15 /*
12 */ && service < 30 /* 13 */) {
      days += 2 /* 14 */
   }
   if(service >= 30 /* 15 */ && age >= 60 /* 16 */) {
      days += 3 /* 17 */
   }

   return days
}
```

The comments mark parts of the code with a number. I'll mutate these predicates or expressions and check what test cases fail. If there is some error in the implementation, the failing test cases indicate that we've caught the mutation. These mutations are off-by-one errors, writing >=, <=, >, < in place of the original operator. These simulate the case, when the programmer makes a mistake during implementation.

In the following tables (Table 19, Table 20, Table 21), we can see that every kind of mutation was caught by at least one test case generated by my GPT, shown in Table 18.

No.	Mutated	Test cases catching it
1	`let days = 21`	all
1	`let days = 23`	all
2	`age <= 18`	#4, #10, #13, #15
2	`age > 18`	#1, #2, #3, #5, #8, #9, #11, #22
2	`age < 17`	#1
2	`age < 19`	#4, #10, #13, #15
3	`age > 60`	#19
3	`age <= 60`	#2, #4, #5, #8, #9, #10, #11, #13, #15, #21, #22
3	`age >= 59`	#11
3	`age >= 61`	#19

Table 19: Mutation predicates and expressions in Paid vacation Days (1)

No.	Mutated	Test cases catching it
4	`service > 30`	#16, #17
4	`service <= 30`	#2, #4, #5, #8, #9, #10, #11, #12, #13, #15, #18, #22
4	`service >= 29`	#2, #4, #9, #11
4	`service >= 31`	#16, #17
5	`days += 4`	#1, #3, #6, #7, #12, #14, #16, #17, #18, #19, #20, #21
5	`days += 6`	#1, #3, #6, #7, #12, #14, #16, #17, #18, #19, #20, #21
6	`age > 45`	#2
6	`age <= 45`	#1, #3, #4, #5, #8, #9, #10, #11, #13, #15, #22
6	`age < 45`	#1, #2, #3, #4, #5, #8, #9, #10, #11, #13, #15, #22
6	`age >= 44`	#8, #9
6	`age >= 46`	#2
7	`age > 60`	#2, #5, #11, #21
7	`age <= 60`	#19
7	`age < 59`	#11
7	`age < 61`	#19
8	`service <= 30`	#17
8	`service > 30`	#2, #5, #11, #18
8	`service < 29`	#2, #11
8	`service < 31`	#17
9	`days += 1`	#2, #5, #11
9	`days += 3`	#2, #5, #11
10	`age > 18`	#4, #10
10	`age <= 18`	#1, #3, #8, #9, #22
10	`age >= 17`	#1
10	`age >= 19`	#4, #10
11	`age <= 45`	#2
11	`age > 45`	#4, #5, #8, #9, #10, #11, #19, #21, #22
11	`age < 44`	#8, #9
11	`age < 46`	#2
12	`service > 15`	#8, #10
12	`service <= 15`	#4, #9, #13, #15, #22
12	`service >= 14`	#13
12	`service >= 16`	#8, #10

Table 20: Mutation predicates and expressions in Paid vacation Days (2)

No.	Mutated	Test cases catching it
13	`service <= 30`	#16
13	`service > 30`	#4, #8, #9, #10, #12, #22
13	`service < 29`	#4, #9
13	`service < 31`	#16
14	`days += 1`	#4, #8, #9, #10, #22
14	`days += 3`	#4, #8, #9, #10, #22
15	`service > 30`	#20
15	`service <= 30`	#14, #19, #21
15	`service >= 29`	#19
15	`service >= 31`	#20
16	`age > 60`	#20
16	`age <= 60`	#6, #7, #12, #14, #16, #17, #18
16	`age >= 59`	#17
16	`age >= 61`	#20
17	`days += 2`	#14, #20
17	`days += 4`	#14, #20

Table 21: Mutation predicates and expressions in Paid vacation Days (3)

6.3 Comparison to BWDM

In a paper about BWDM [29], there are two example functions, which I can test in GPT as well. They are written in VDM++, but the if structure can be converted to GPT Lang.

Evaluate Grades

The code can be found in Appendix A.6.

- BWDM's BVA had generated 14 test cases.
- My GPT with MONKE generated 15 test cases.

Quarter

The code can be found in Appendix A.7.

- BWDM's BVA had generated 12 test cases.
- My GPT with MONKE generated 11.

Their paper didn't list all of the test cases, so I couldn't compare the actual test values. But, my GPT implementations generates around the same number of test cases. These are small functions, but in the future it'd be worth to test more complex functions, where there are orders of magnitude more test cases, to see if the GPT algorithm combined with graph reduction makes a difference.

7 Coding in Rust

I used the latest Rust version for this project, which is 1.69[1] at the time of writing.

Rust is a performant high-level programming language [30]. It borrows functional techniques from other languages, like monadic error handling and a rich type system. Instead of garbage collectors, it uses borrow checking to solve memory management.

The Rust compiler can catch a lot of errors during compilation. Its rich typesystem, and lint tools like clippy[2], helps writing correct code. For GPT, this was crucial, as there must not be any errors in test case generation.

Another advantage of Rust is that it uses LLVM[3] to compile to native code. Combined with no garbage collection, this allows Rust code to be faster than other languages [31].

Rust can also compile to WebAssembly[4], which makes the code available in browsers on websites. This means, that GPT can be run on the web, making the test generation tool more easily available for test designers, they don't even have to install the program on their computer.

The source code of my GPT implementation is open-source and available on GitHub at https://github.com/test-design-org/general-predicate-testing. The code consist of three main modules:

- `gpt-common`: This hosts the main logic of GPT: parsing, AST, IR, test case generation with GPT, graph reduction.
- `gpt-cli`: This is the command line tool that can generate test cases from `.gpt` source files. It has configurable options, like what graph reduction Algorithm to use, or in what format to display the generated test cases.
- `gpt-frontend`: This contains the frontend application, that compiles the Rust code to HTML and Javascript, so that it can be opened in a web browser.

I'd like to highlight some of the core libraries I used:

- **nom**[5]: Parser combinator library for writing the GPT Lang parser.
- **petgraph**[6]: Graph library with some common graph algorithms included.
- **clap**[7]: Command Line Argument Parser for `gpt-cli`.
- **yew**[8]: Front end framework for creating web applications in Rust.

[1] https://blog.rust-lang.org/2023/04/20/Rust-1.69.0.html
[2] https://github.com/rust-lang/rust-clippy
[3] https://rustc-dev-guide.rust-lang.org/overview.html
[4] https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm
[5] https://github.com/rust-bakery/nom
[6] https://github.com/petgraph/petgraph
[7] https://github.com/clap-rs/clap
[8] https://github.com/yewstack/yew

8 Summary

As I've shown, my GPT implementation works according the book [2]. I've validated that the automatically generated and reduced test cases cover all kinds of predicate errors.

This tool can be used by test designers to automate the test generation process and validate critical systems. Because of it's automated nature, it can handle problems that were too complex for humans to do by hand. For example, `complex_hard.gpt` in Section 5.6.5, which had over 1.2 million possible intersections between the NTuples before graph reduction.

I could also extend the original GPT algorithm with the handling of disjunctions. This is an important feature, as it allows test designers to describe the requirements closer to the specification. They don't have to come up with the correct equivalence partitions and conjunctive forms, which is also a lot of manual work and prone to error.

Graph reduction had great results, with the most complex example having a 90.32% reduction of test cases. This means, that it is enough to use only a tenth of the number of test cases, but we still get the same test coverage. This can help test designers, who have to calculate the outputs for all of these test cases according to the specifications. It also reduces the amount of time it takes for the unit tests to run, improving the efficiency of Continuous Integration systems.

In their paper, Murane argues that "One disadvantage with boundary value analysis is that it is not as systematic as other prescriptive testing techniques" [5]. As I've demonstrated, GPT is systematic and test cases can be generated automatically with my GPT implementation.

I think that this tool can be foundation for the GPT technique, and it can be used as an example of the effectiveness of automatic black-box BVA test generation.

8.1 Acknowledgements

I'd like to thank Kovács Attila for his mentorship and guidance for almost two years during my studies. Without him and his work on General Predicate Testing, I couldn't have achieved these results.

I'd also like to thank the original group with whom I've started this GPT journey. We've brainstormed about the first implementations of GPT during our Requirement Analysis course, and we came up with MONKE. It's so wonderful to see that MONKE is still going strong after all these years.

Last, but not least, I'd like to thank Gyimesi Kristóf for his dedicated support during all four semesters, his great insights when designing complex algorithms, and the massive amount of time he put into proofreading this thesis.

9 Future improvement ideas

9.1 Coincidental Correctness

In their paper, Hierons has shown that because BVA only generates single points on boundaries, geometric shifts in boundaries can lead to accidental correctness [32]. In other words, there are some incorrect implementations, which cannot be caught with BVA.

Consider the following example:
Write a program that accepts a point as an (x, y) coordinate, where x and y are integers. Return true, if the point is above the $f(x) = 0$ function's slope, otherwise return false.

From this, we'd write it in GPT Lang as:

```
var x: int
var y: int
if(y > 0) else
```

You can notice, that we don't reference x here, so the generated test cases won't care for x either. If we default x to 0 in all test cases, we'd have the following reduced generated test cases: `A = { x: 0, y: 1 }, B = { x: 0, y: (-Inf, -1] }, C = { x: 0, y : [2, Inf) }, D = { x: 0, y: 0 }`

The problem with this is that if the implementation checked against the $f(x) = x$ function's slope, all of our test cases would still pass. This is, because in the $x = 0$ point both $f(x) = 0$ and $f(x) = x$ behave in the same way. However, we know that checking against $f(x) = x$ would be an incorrect implementation.

Figure 9 shows the two functions: $f(x) = 0$ in green and $f(x) = x$ in orange. In this scenario the test points A, B, C, and D show that both functions behave in the same way. In contract, point E and F could show the bug, because they are both under the slope of $f(x) = x$.

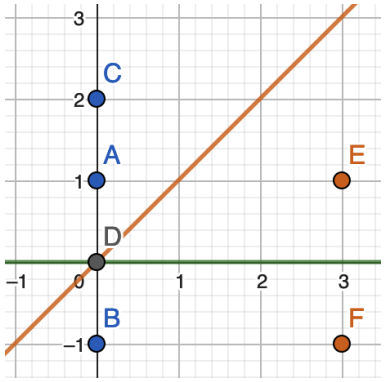

Figure 9: Coincidental Correctness example

9.2 Linear Predicates

Currently, GPT can only handle predicates with one variable. Linear predicates are not supported.

Example: $i < j + 1$

9.3 Different equivalence partitions for implementations

When implementing a program, if we differ from the specification by some margin, we might unknowingly create an implementation that has different equivalence partitions.

Example:

Paid Vacation Days reference implementation:

```
function paidVacationDays(age, service) {
   let days = 22

   if (age < 18 || age >= 60 || service >= 30) {
      days += 5
   }
   if(age >= 45 && age < 60 && service < 30) {
      days += 2
   }
   if(age >= 18 && age < 45 && service >= 15 && service < 30) {
      days += 2
   }
   if(service >= 30 && age >= 60) {
      days += 3
   }

   return days
}
```

Paid vacation days different implementation:

```
function paidVacationDays(age, service) {
   let days = 22

   if (age < 18 || age >= 60 || service >= 30) {
      days += 5

      if (age >= 60 && service >= 30) {
         days += 3
      }
   } else if (service >= 15 || age >= 45 /* <- This */) {
      days += 2
   }

   return days
}
```

In the second implementation, if we replace that `age >= 45` with any number in $[46, 59]$ the original tests in Table 18 still all pass. However, we know for sure that we've made an error, because we've replaced that number.

An example test case for this error would be `{age: [46, 59], service: (-Inf, 14]}`.

This is, because this implementation has different equivalence partitions. In the original tests we only had to test M49 to test that service goes up to 28, not 14. Because of graph reduction and how we select test points, it's possible that we select a number from $[15, 28]$. In this implementation with the `||`, this test case will only cover the left-hand side of the `||`, because it sees a test case with `service >= 15` and it won't test the `age` part.

Because we've differed from the equivalence partitions that we've defined our GPT Lang definition with, the way we can solve this is to create another GPT Lang definition for this implementation. This way, we'll have a test case that covers the `service >= 15 || age >= 45` condition, which didn't exist in the original one.

This is related to the Competent Programmer Hypothesis defined in Section 1.3, because we don't want to test implementations that are vastly different or more overcomplicated than a simple solution. In this case, the `else if`, the nested `if` and additional `||` made this implementation more complex.

9.4 Supporting enums in GPT Lang

Enums could be modelled like booleans for BVA. The only operators we need to support is == and `!=`. For `!=`, we can create a list of all the other enum values. When intersecting NTuples, we intersect the sets of these possible enum values.

Enums would let us model some behaviors. For example, when we have a function that does a database query, we could model it as `SUCCESS | TIMEOUT | CONNECTION_ERROR`. This is basically equivalence partitioning the results of any functions.

9.5 Supporting Strings

There has been research about extending BVA to strings [33]. This approach could be investigated further, as it would let us use GPT in even more situations. Handling strings is an indispensable part of programs, so having an automatic test generation solution for them could be very useful.

9.6 Hierarchical GPT

Hierarchical GPT is a concept, where our requirements have sequential parts, hierarchies. There could be a first part which uses the variables `a, b, c` and a second part which uses `c, d, e`. If we were to generate test cases with `a, b, c, d, e` our initial graph would have a lot of edges.

The idea is, that these separate parts can be generated independently of each other, reduced independently, and the test cases would be joined in the end. This would reduce the complexity of graph reduction and avoid the combinatorial explosion.

A. Appendix

A.1 Planned Vacation Days

```
/*
Note: requirements can be specified and included in comments of GPT
Lang files.

Paid vacation days

R1 The number of paid vacation days depends on age and years of
service. Every employee receives at least 22 days per year.

Additional days are provided according to the following criteria:
R2-1 Employees younger than 18 or at least 60 years, or employees
with at least 30 years of service will receive 5 extra days.
R2-2 Employees of age 60 or more with at least 30 years of service
receive 3 extra days, on top of possible additional days already given
based on R2-1.
R2-3 If an employee has at least 15 years of service, 2 extra days
are given. These two days are also provided for employees of age 45
or more. These extra days cannot be combined with the other extra
days.

Display the vacation days. The ages are integers and calculated with
respect to the last day of December of the current year.
*/

var age: int
var service: int

// R1
if(age < 18 && service < 30)
if(age >= 60 && service < 30)
if(service >= 30 && age < 60 && age >= 18)

// R1-2
if(service >= 30 && age >= 60)

// R1-3
if(service >= 15 && age < 45 && age >= 18 && service < 30)
if(age >= 45 && service < 30 && age < 60)
```

A.2 Price Calculation

```
/*
Price calculation
R1 The customer gets 10% price reduction if the price of the goods
reaches 200 euros.
R2 The delivery is free if the weight of the goods is under 5
kilograms.
    Reaching 5 kg, the delivery price is the weight in euros, thus,
when the products together are 6 kilograms then the delivery price is
6 euros.
    However, the delivery remains free if the price of the goods
exceeds 100 euros.
R3 If the customer prepays with a credit card, then s/he gets 3%
price reduction from the (possibly reduced) price of the goods.
R4 The output is the price to be paid. The minimum price difference
is 0.1 euro, the minimum weight difference is 0.1 kg.
*/

var prepaid_with_credit_card: bool
var price: num(0.1)
var weight: num(0.1)

// R1
if(price >= 200)

// R2
if(weight >= 5 && price <= 100)

// R3
if(prepaid_with_credit_card == true)
```

A.3 complex_small.gpt

```
var x: num
var y: num
var z: num

if(x != 1 || y != 1 || z != 1)
else if(x != 2 || y != 2 || z != 2)
else

if(x != 10 || y != 20)
else if(x != 2)
else
```

A.4 complex_medium.gpt

```
var x: num
var y: num
var z: num

if(x != 1 || y != 1 || z != 1)
else if(x != 2 || y != 2 || z != 2)
else

if(x != 10 || y != 20)
else if(x != 2)
else if(z == 3)
else
```

A.5 complex_hard.gpt

```
var x: num
var y: num
var z: num

if(x != 1 || y != 1 || z != 1)
else if(x != 2 || y != 2 || z != 2)
else if(x != 3 || y != 3 || z != 3)
else


if(x != 10 || y != 20)
else if(x != 2)
else if(x != 3 && y != 3 && z != 3)
else if(x != 4 && y != 4 && z != 4)
else
```

A.6 Evaluate Grades

```
/*
Original requirements in VDM++:

class GradeEvaluation
   functions
   evaluateGrades : nat -> seq of char
   evaluateGrades (point) ==
      if(point >= 60) then
         if(point >= 70) then
            if(point >= 80) then
               if(point >= 90) then
                  "A"
               else
                  "B"
            else
               "C"
         else
            "D"
      else
         "F"
   pre point <= 100;
end GradeEvaluatio
*/

var point: int

if(point <= 100)

if(point >= 60) {
   if(point >= 70) {
      if(point >= 80) {
         if(point >= 90) {
            // "A"
         } else
            // "B"
      } else
         // "C"
   } else
      // "D"
} else
   // "F"
```

A.7 Quarter

```
/*
Original requirements in VDM++:

class Quarter
types
   public MONTH = nat1
   inv m == m <= 12;
functions
   determineQuarter : MONTH -> seq of char
   determineQuarter (month) ==
      if(month <= 3) then
         "Q1"
      else
         if(month <= 6) then
            "Q2"
         else
            if(month <= 9) then
               "Q3"
            else
               "Q4";
end Quarter
*/
var month: int

if(month <= 12)

if(month <= 3)
   // "Q1"
else if(month <= 6)
   // "Q2"
else if(month <= 9)
   // "Q3"
else
   // "Q4";
```

Bibliography

[1] "CNN - Metric mishap caused loss of NASA orbiter - September 30, 1999," 2023. [Online]. Available: http://edition.cnn.com/TECH/space/9909/30/mars.metric.02

[2] I. Forgacs, and A. Kovacs, *Paradigm Shift in Software Testing: Practical Guide for Developers and Testers*, Independently Published, 2022.

[3] M. E. Khan, and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl.*, vol. 3, no. 6, 2012.

[4] A. Mustafa, W. M. Wan-Kadir, et al., "Automated test case generation from requirements: a systematic literature review," *Computers, Materials Continua*, vol. 67, no. 2, pp. 1819–1833, 2021.

[5] T. Murnane, and K. Reed, "On the effectiveness of mutation analysis as a black box testing technique," in *Proc. 2001 Australian Softw. Eng. Conf.*, 2001, pp. 12–20.

[6] S. Nidhra, and J. Dondeti, "Black box and white box testing techniques-a literature review," *Int. J. Embedded Syst. Appl. (Ijesa)*, vol. 2, no. 2, pp. 29–50, 2012.

[7] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *Proc. 2nd Int. Workshop Random Testing: Co-Located 22nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE 2007)*, 2007, p. 1.

[8] P. Godefroid, M. Y. Levin, D. A. Molnar, and others, "Automated whitebox fuzz testing.," in *Ndss*, vol. 8, 2008, pp. 151–166.

[9] H. S. Son, R. Y. C. Kim, and Y. B. Park, "Test case generation from cause-effect graph based on model transformation," in *2014 Int. Conf. Inf. Sci. & Appl. (Icisa)*, 2014, pp. 1–4.

[10] C. R. Rao, "Orthogonal arrays," *Scholarpedia*, vol. 4, no. 7, p. 9076, Jul. 2009, doi: 10.4249/scholarpedia.9076.

[11] B. Beizer, *Software Testing Techniques*, New York : Van Nostrand Reinhold, 1983.

[12] R. V. Binder, *Testing Object-Oriented Systems*, Boston, MA: Addison-Wesley Educational, 1999.

[13] I. Forgacs, and A. Kovacs, *Practical Test Design*, Swindon, England: BCS, The Chartered Institute for IT, 2019.

[14] L. Rigby, P. Denny, and A. Luxton-Reilly, "A miss is as good as a mile: off-by-one errors and arrays in an introductory programming course," in *Proc. Twenty-Second Australas. Comput. Educ. Conf.*, 2020, pp. 31–38.

[15] G. Fraser, and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, 2013, doi: 10.1109/TSE.2012.14.

[16] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?," in *Proc. 2013 Int. Symp. Softw. Testing Anal.* in Issta 2013, Lugano, Switzerland, 2013, p. 291, doi: 10.1145/2483760.2483774. [Online]. Available: https://doi.org/10.1145/2483760.2483774

[17] Z. Zhang, T. Wu, and J. Zhang, "Boundary value analysis in automatic white-box test generation," in *2015 IEEE 26th Int. Symp. Softw. Rel. Eng. (Issre)*, vol. 0, 2015, pp. 239–249, doi: 10.1109/ISSRE.2015.7381817.

[18] V. Ambriola, and V. Gervasi, "On the systematic analysis of natural language requirements with c irce," *Automated Softw. Eng.*, vol. 13, pp. 107–167, 2006.

[19] R. Azamfirei, S. R. Kudchadkar, and J. Fackler, "Large language models and the perils of their hallucinations," *Crit. Care*, vol. 27, no. 1, pp. 1–2, 2023.

[20] P. Samuel, and R. Mall, "Boundary value testing based on uml models," in *14th Asian Test Symp. (ATS'05)*, 2005, pp. 94–99.

[21] P. Mani, and M. Prasanna, "Test case generation for embedded system software using uml interaction diagram," *J. Eng. Sci. Technol.*, vol. 12, no. 4, pp. 860–874, 2017.

[22] T. Katayama, H. Tachiyama, et al., "Bwdm: test cases automatic generation tool based on boundary value analysis with vdm++," *J. Robotics, Netw. Artif. Life*, vol. 4, p. 110, 2017, doi: 10.2991/jrnal.2017.4.2.1.

[23] T. Katayama, F. Hirakoba, et al., "Application of pairwise testing into bwdm which is a test case generation tool for the vdm++ specification," *J. Robotics, Netw. Artif. Life*, vol. 6, no. 3, pp. 143–147, 2019, doi: 10.2991/jrnal.k.191202.001. [Online]. Available: https://doi.org/10.2991/jrnal.k.191202.001

[24] "intervals-general - crates.io: Rust Package Registry," 2023. [Online]. Available: https://crates.io/crates/intervals-general

[25] AlgoDaily, "AlgoDaily - Using the Two Pointer Technique - Introduction," 2023. [Online]. Available: https://algodaily.com/lessons/using-the-two-pointer-technique

[26] dkaeae (https://cs.stackexchange.com/users/70382/dkaeae), "Given a set of intervals on the real line, find a minimum set of points that 'cover' all the intervals." [Online]. Available: https://cs.stackexchange.com/q/101911 (Computer Science Stack Exchange)

[27] cmyr, "cargo-instruments," 2023. [Online]. Available: https://github.com/cmyr/cargo-instruments

[28] "Instruments Help," 2023. [Online]. Available: https://help.apple.com/instruments/mac/10.0/#

[29] T. Muto, T. Katayama, et al., "Expansion of application scope and addition of a function for operations into bwdm which is an automatic test cases generation

tool for vdm++ specification," *J. Robotics, Netw. Artif. Life*, vol. 9, no. 3, pp. 255–262, 2022.

[30] "Rust Programming Language," 2023. [Online]. Available: https://www.rust-lang.org/

[31] W. Bugden, and A. Alahmar, "Rust: the programming language for safety and performance," *Arxiv Preprint Arxiv:2206.05503*, 2022.

[32] R. M. Hierons, "Avoiding coincidental correctness in boundary value analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, p. 227, Jul. 2006, doi: 10.1145/1151695.1151696. [Online]. Available: https://doi.org/10.1145/1151695.1151696

[33] A. Jain, S. Sharma, S. Sharma, and D. Juneja, "Boundary value analysis for non-numerical variables: strings," *Orient. J. Comp. Sci. Technol*, vol. 3, no. 2, 2010.