Appendix A

Implementation

In the following section, the important parts of our implementation of the group sampling algorithm are included. The Implementation is written in python and uses common packages out of its ecosystem. Most notably **sklearn** for regression methods, **pandas** and **numpy** for data manipulation, **networkx** for graph related tasks and **z3-solver** as a wrapper for python to use the Z3 Theorem prover.

A.1 Sampling Stratgies

```
import logging
import time

import z3
from z3 import And, Not

from model.variability_model import VariabilityModel
from sampling.group_sampling.group_sampling_strategy import GroupSamplingStrategy
from util.util import flatten


class HammingGroupSamplingStrategy(GroupSamplingStrategy):
    _ctx = "[HammingGroupSamplingStrategy]"

    def __init__(self, vm: VariabilityModel, **opts):
        super().__init__(vm, **opts)
        self.all_first_groupings = []
        self.solutions = []

        self.solver = z3.Optimize()
        self.solver.set("timeout", 60000)
        self.solver.add(self.vm.create_z3_constrains(shuffle=True))
        self.solver.push()

    def reset(self):
        '''
        Helper function to reset the sampler.
        '''
        self.solutions = []
        self.all_first_groupings = []

    def _optimize_for_hamming_between_groups(self, i, groups):
        if i == 1:
            self.all_first_groupings.append(groups[i - 1])
        if i == 0:
            group_distance_funcs = [
                z3.Sum([
                    z3.If(z3.Bool(var.name()), 0, 1)
```

```
                    if z3.is_true(group[var]) else 0
                    for var in group
                ])
                for group in self.all_first_groupings
            ]
            for func in group_distance_funcs:
                self.solver.maximize(func)

    def _set_preferred_group_size(self):
        # Set group sizes
        feature_per_group = len(self.vm.get_features()) // self.group_size
        sum_features = z3.Sum(
            [z3.If(z3.Bool(feature), 1, 0) for feature in self.vm.get_features()]
        )
        # We set the group size as a soft constraint with a high weighting
        # This is done because z3 sometimes tries too hard to find a solution
        # fitting the constraint and the hamming distance and takes way too
        # long to return a result. If the soft constraint is set, it, in theory,
        # can break the constraint which lets it better compute an optimal
        # solution for the hamming distance. Breaking the constraint very rarely happens.
        self.solver.add_soft(sum_features == feature_per_group, weight=10)

    def _add_hamming_cost_function(self, groups):
        # If no other groups are created, we can't optimize the distance
        if len(groups) == 0:
            return

        distance_per_feature_per_group = [
            [
                z3.If(z3.Bool(var.name()) == z3.is_true(group[var]), 1, 0)
                for var in group
            ]
            for group in groups
        ]
        distance_all_features_all_groups = flatten(distance_per_feature_per_group)
        hamming_cost_function = z3.Sum(distance_all_features_all_groups)
        self.solver.maximize(hamming_cost_function)

    def get_sample(self):
        groups = []
        logging.debug(f"{self._ctx} Get sample with groups size {self.group_size}")
        # We want to keep track, how long the generation of a group takes
        t0_grouping = time.time()

        # Save the current state of the constraints in the solver
        self.solver.push()
        for i in range(0, self.group_size):
            logging.debug(f"{self._ctx} Searching for group {i}")
            # Keep track of when the sampling for the group started
            t0_group = time.time()

            self._set_preferred_group_size()

            self._optimize_for_hamming_between_groups(self.solver, i, groups)

            self._add_hamming_cost_function(groups)

            if self.solver.check() == z3.sat:
                logging.debug(f"{self._ctx} Group generation {time.time() - t0_group}s")
                # We retrieve the solution Z3 found
                model = self.solver.model()
                # Restore the constraint state of the solver before group generation
                self.solver.pop()
                # Add found model to the constraints to avoid getting the same model again
                self.solver.add([Not(And([v() == model[v] for v in model]))])
                # Save the current state of the solver so that the found model is kept as
                # constraint for the next run
                self.solver.push()
                groups.append(model)
            else:
                raise Exception('No more samples can be found!')
        logging.debug(f"{self._ctx} Grouping generation: {time.time() - t0_grouping}s")
        return [self.transform_model(m) for m in groups]
```

Strategy A.1: Hamming distance based group sampling stratgey

```
import logging
import random
import time

import numpy as np
import z3
```

```python
from z3 import And, Not

from model.variability_model import VariabilityModel
from sampling.group_sampling.group_sampling_strategy import GroupSamplingStrategy
from util.network_util import find_components, load_mutex_graph, \
    generate_mutex_graph_squential, find_optional_features
from util.util import flatten, distribution_with_rest


class IndependentFeatureGroupSamplingStrategy(GroupSamplingStrategy):
    _ctx = "[IndependentFeatureGroupSamplingStrategy]"

    def __init__(self, vm: VariabilityModel, **opts):
        super().__init__(vm, **opts)
        self.solutions = []
        self.all_groupings = []

        if opts.get('load_mutex'):
            logging.debug(f'{self._ctx} Loading mutex graph: {opts.get("load_mutex")}')
            self.mutex_graph = load_mutex_graph(opts.get('load_mutex'))
        else:
            logging.debug(f'{self._ctx} Generating mutex graph')
            self.mutex_graph = generate_mutex_graph_squential(vm)

        self.mutex_components = find_components(self.mutex_graph)
        self.optionals = find_optional_features(self.vm)
        self.independent_features = self.optionals - set(flatten(self.mutex_components))

        self.solver = z3.Optimize()
        self._add_constrains()
        self.solver.set("timeout", 60000)

        logging.debug(f'{self._ctx} Feature model Information: ------')
        logging.debug(f'{self._ctx} Mutexes: {self.mutex_components}')
        logging.debug(f'{self._ctx} Optional features: {self.optionals}')
        logging.debug(f'{self._ctx} Independent features: {self.independent_features} \n')

    def reset(self):
        """
        Helper function to reset the sampler
        """
        self.solutions = []
        self.all_groupings = []

    def _add_constrains(self):
        """
        Add the constraints of the feature model to the solver
        """
        self.solver.add(self.vm.create_z3_constrains(shuffle=True))
        random.shuffle(self.solutions)
        for solution in self.solutions:
            self.solver.add(solution)

    def _independent_features_per_group(self, i):
        """
        Adds the amount of independent features as a constraint
        """
        # We want all independent features to be in a group.
        # This is not always possible with groups of the same size
        # We create a distribution of parameters of equal size with the last group
        # acting as overflow for features which do not fit in another group anymore
        group_sizes = distribution_with_rest(self.independent_features, self.group_size)
        sum_independent_features = z3.Sum([
            z3.If(z3.Bool(var), 1, 0)
            for var in self.independent_features
        ])
        # We set the amount of independent features as soft constraint to avoid
        # long runtimes of z3 when optimizing for a cost function
        self.solver.add_soft(sum_independent_features == group_sizes[i], weight=10)

    def _pick_random_mutex_feature(self):
        for idx, mutex in enumerate(self.mutex_components):
            mutex_feature = np.random.choice(mutex)
            self.solver.add(z3.Bool(mutex_feature))

    def _optimize_for_hamming_between_groups(self, i, groups):
        if i == 1:
            self.all_groupings.append(groups[i - 1])
        if i == 0:
            group_distance_funcs = [
                z3.Sum([
                    z3.If(z3.Bool(var.name()), 0, 1)
                    for var in group
                    if var.name() in self.independent_features and z3.is_true(group[var])
```

```
                ])
                for group in self.all_groupings
            ]
            for func in group_distance_funcs:
                self.solver.maximize(func)

    def _no_overlap_in_independent_features(self, groups):
        if len(groups) != 0:
            distances = [[
                z3.If(z3.Bool(var.name()), 1, 0)
                if var.name() in self.independent_features and z3.is_true(group[var])
                else 0
                for var in group
            ] for group in groups]
            distance_func = z3.Sum(flatten(distances))
            self.solver.add(distance_func == 0)

    def get_sample(self):
        groups = []
        logging.debug(f"{self._ctx} Get sample with groups size {self.group_size}")
        # We want to keep track, how long the generation of a group takes
        t0_grouping = time.time()

        # Save the current state of the constraints in the solver
        self.solver.push()
        for i in range(0, self.group_size):
            logging.debug(f"{self._ctx}  Searching for group {i}")
            # Keep track of when the sampling for the group started
            t0_group = time.time()

            self._independent_features_per_group(i)

            self._pick_random_mutex_feature()

            self._no_overlap_in_independent_features(groups)

            self._optimize_for_hamming_between_groups(i, groups)

            if self.solver.check() == z3.sat:
                logging.debug(f"{self._ctx} Group generation: {time.time() - t0_group}s")

                model = self.solver.model()
                self.solver.pop()
                self.solver.add([Not(And([v() == model[v] for v in model]))])
                self.solver.push()
                groups.append(model)
            else:
                raise Exception('No more samples can be found!')

        logging.debug(f"{self._ctx}  Grouping generation: {time.time() - t0_grouping}s")
        return [self.transform_model(m) for m in groups]
```

Strategy A.2: Group sampling stratgey with independent features

A.2 Graph algorithms

```
import itertools
import logging
import time

import networkx as nx
from z3 import z3

def generate_mutually_exclusive_feature_graph(vm):
    logging.debug('Start generating graph')
    # We want to know how long it takes
    t0 = time.time()

    mutex_graph = nx.Graph()
    result = []
    solver = z3.Solver()
    solver.add(vm.create_z3_constrains())

    # Iterate over each possible combination of two features
    for i, j in itertools.combinations(vm.get_features(), 2):
        # Save the current solver constraints
        solver.push()
```

```
        # Add a constraint where both are enabled
        constraint_i = z3.And([
            z3.Bool(i), z3.Bool(j)
        ])
        solver.add(constraint_i)

        if solver.check() == z3.unsat:
            # If it is not satisfiable, the
            # featues are mutually exclusive
            result.append([i, j])
        solver.pop()

    for edge in result:
        i, j = edge
        mutex_graph.add_edge(i, j)

    logging.debug(f'Time {time.time() - t0}')
    return mutex_graph
```

Algorithm A.3: Generation of the mutually exclusive feature graph

```
from z3 import z3

def find_true_optional_features(vm):
    optionals = []
    constrains = vm.create_z3_constrains()
    for feature in vm.get_features():
        solver = z3.Solver()
        solver.add(constrains)

        # Add constraint to disable a feature
        solver.add(z3.Not(z3.Bool(feature)))

        if solver.check() == z3.sat:
            # If it is satisfiable, it is
            # a true optional feature
            optionals.append(feature)

    return optionals
```

Algorithm A.4: Find true optional features

A.3 Group Sampling - Learning

```
import logging
import time
import warnings
from typing import List

import numpy as np
import pandas
from numba import jit
from sklearn import linear_model

from model.variability_model import VariabilityModel
from util.util import flatten

warnings.simplefilter(action='ignore', category=FutureWarning)


class Berta:
    _ctx = "[Group Sampling Model]"

    def __init__(self, vm: VariabilityModel, group_size):
        self.grouping_influences = None
        self.model = None
        self.coefficients = None
        self.group_size = group_size
        self.vm = vm

    def fit(self, x: List[List[List[int]]], y: List[List[int]]):
        '''
            Distribution  Result  Features      Feature Influence/Sensitivity
            1 0 0         10      1 0 1 1 0 1   10   None 10   10   None 10
```

```
            0 1 0          11       1 1 1 0 0 1    11    11    11    None None 11
            0 0 1          12       1 0 0 1 1 1    12    None None 12    12    12
            1 0 0          12       1 1 0 0 1 1    12    12    None None 12    12
            [...]
            ----------------------------------------------------------------------
                                    Average:       11    11    10,5 11    12    11
                                                   ------------------------------
                         Most influential:                               12
        '''
        logging.debug(f"{self._ctx} Fitting data")
        t0 = time.time()

        grouping_influences = pandas.DataFrame()
        intercepts = []
        # Iterate over each grouping which got measured
        for idx, groupings in enumerate(x):
            # We create an identity matrix to fit our
            # linear regression on.
            distribution = np.identity(self.group_size)
            # regression = linear_model.Lasso()
            regression = linear_model.LinearRegression(fit_intercept=True)
            # Fit the results of each group with the identity matrix
            regression.fit(distribution, y[idx])
            intercepts.append(regression.intercept_)
            group_result = pandas.DataFrame()
            for key, dist in enumerate(distribution):
                group = groupings[key]
                influence = regression.coef_[key]
                expanded = [[influence if feature == 1 else None for feature in group]]
                group_result = group_result.append(expanded)

            grouping_influences = grouping_influences.append(
                group_result.astype("float64"),
                ignore_index=True
            )

        groupings = pandas.DataFrame(flatten(x))

        logging.debug(f"{self._ctx} Fitting group influences took {time.time() - t0}.")
        t1 = time.time()
        self.grouping_influences = grouping_influences

        grouping_influences[0] = np.nan
        groupings[0] = np.nan

        c = pandas.DataFrame(columns=groupings.columns)
        for i in range(0, len(groupings.columns)):
            max_influence, indices = self.get_max_influence(grouping_influences)
            if len(indices) < 5:
                if i == 0:
                    logging.info(
                        f"{self._ctx}  Most influential parameter {indices + 1}"
                    )
                else:
                    logging.debug(
                        f"{self._ctx} {i + 1}th influential parameter: {indices + 1}"
                    )

            for index in indices:
                c.loc[0, index] = max_influence
                grouping_influences = pandas.DataFrame(
                    nb_where(
                        grouping_influences.to_numpy(),
                        groupings.to_numpy(),
                        index, max_influence
                    )
                )
                grouping_influences.loc[:, index] = np.NaN

        c = c.fillna(0)
        self.coefficients = c
        self.model = linear_model.LinearRegression()
        self.model.intercept_ = np.mean(intercepts)
        self.model.coef_ = c.to_numpy()

        logging.debug(f"{self._ctx} Stepwise analysis took {time.time() - t1}.")
        logging.info(f"{self._ctx} Fitting data took {time.time() - t0}.")

    def get_max_influence(self, grouping_influences):
        mean_influences = grouping_influences.mean()
        mean_mean_influences = mean_influences.mean()
        mean_influences = mean_influences - mean_mean_influences
        max_idx = mean_influences.abs().idxmax()
        if max_idx != max_idx:
```

```
            return None, []
        max_influence = mean_influences[max_idx]
        influences = mean_influences[mean_influences == max_influence]
        indices = influences.index
        return (max_influence + mean_mean_influences) / len(indices), indices

    def predict(self, x: List[List[int]]):
        return self.model.predict(x)


@jit(nopython=True)
def nb_where(influences, groupings, index, sub):
    for row in range(len(influences)):
        for col in range(len(influences[row])):
            if groupings[row][index] == 1:
                influences[row][col] = influences[row][col] - sub
    return influences
```

Algorithm A.5: Generate model