Failed Cases Report

This document serves as the detailed failed cases report for Research Question 4 (RQ4) of the paper *TransLibEval: A Benchmark for Evaluating Code Translation with Third-Party Libraries*. The primary purpose of this appendix is to provide a comprehensive taxonomy and concrete examples of the various failure patterns encountered during our extensive evaluation of state-of-the-art code translation systems when handling third-party library usage across programming languages.

Our analysis reveals that third-party library integration presents unique and previously underexplored challenges for automated code translation. While existing benchmarks primarily focus on syntactic and basic semantic equivalence, they often overlook the complex ecosystem dependencies that arise in real-world software development. It is important to note that due to the complexity and verbosity of the actual programs in our dataset, the examples presented in this appendix are specifically crafted for clarity and pedagogical purposes. While they accurately represent the core failure patterns observed in our study, they are simplified illustrations rather than direct excerpts from the benchmark to facilitate reader comprehension.

This appendix systematically categorizes these failure modes, providing researchers and practitioners with: (1) a structured framework for understanding library-related translation errors, (2) clear, illustrative examples demonstrating each failure pattern, and (3) insights into the root causes of these translation failures.

Contents

A Third-Party Reference Errors

A-1 Library-Related Errors

A-1-1 Calling an Invalid Library

Description: This error occurs when the model references a library that does not exist in the target language's ecosystem.

Example:

```
import polars as pl
import numpy as np

def process_large_dataset():
    # Create a large DataFrame with Polars
    df = pl.DataFrame({
        ’id’: range(1000000),
        ’value’: np.random.randn(1000000)
    })

    # Perform complex operations
    result = df.filter(pl.col(’value’) > 0)\
               .groupby(’id’ % 10)\
               .agg(pl.mean(’value’).alias(’avg_value’))

    return result
```

Listing 1: Source Python code using Polars

```
import polars.*;  // Invalid: Polars doesn’t exist in Java
import numpy.*;   // Invalid: NumPy doesn’t exist in Java

public class DataProcessor {
    public static Object processLargeDataset() {
        // Attempt to use Python libraries in Java
        DataFrame df = new DataFrame();  // Invalid
        df.addColumn("id", IntStream.range(0, 1000000).toArray());
        df.addColumn("value", new Random().doubles(1000000).toArray());

        // Invalid Polars-like operations
        DataFrame filtered = df.filter(col("value").gt(0));  // Invalid
        DataFrame result = filtered.groupby("id % 10")
                                  .agg(mean("value").alias("avg_value"));  // Invalid
        return result;
    }
}
```

Listing 2: Translated Java code with invalid Polars reference

Explanation: The model incorrectly attempts to use Python's Polars and NumPy libraries in Java code. Polars is a Python/Rust-based DataFrame library with no Java equivalent in the provided list. The model should have used Java's Tablesaw (`tech.tablesaw:tablesaw-core`) instead of trying to import non-existent Java versions of Python libraries.

A-1-2 Missing Import Statements

Description: This error occurs when the model uses third-party libraries but fails to include the necessary import statements.

Example:

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import com.google.common.collect.Lists;
import org.apache.commons.lang3.StringUtils;

public class JsonProcessor {
    public static String processJson(String input) throws Exception {
        ObjectMapper mapper = new ObjectMapper();
        Object data = mapper.readValue(input, Object.class);

        // Use Guava for list operations
        List<Object> items = Lists.newArrayList(data);

        // Use Apache Commons for string manipulation
        return StringUtils.join(items, "-");
    }
}
```

Listing 3: Source Java code using Jackson and Guava

```python
def process_json(input):
    mapper = ObjectMapper()  # Missing import
    data = mapper.read_value(input, Object)  # Missing import

    # Missing Guava equivalent imports
    items = list(data)  # Basic list conversion instead of Guava equivalent

    # Missing Apache Commons equivalent
    return "-".join(str(item) for item in items)  # Manual join
```

Listing 4: Translated Python code missing imports

Explanation: The model correctly maps the functionality but fails to include the necessary Python import statements. For Jackson's ObjectMapper, it should use `import json` or a more advanced library like Marshmallow. For Guava's Lists utilities, it could use `boltons` or other collection utilities. For Apache Commons StringUtils, it should use Python's built-in string methods or additional libraries.

A-1-3 Retaining Original PL's Library

Description: This error occurs when the model keeps the source language's library imports in the translated code.

Example:

```cpp
#include <boost/algorithm/string.hpp>
#include <boost/lexical_cast.hpp>
#include <nlohmann/json.hpp>

std::string processData(const std::string& input) {
    // Use Boost string algorithms
    std::string trimmed = boost::trim_copy(input);
    std::vector<std::string> parts;
    boost::split(parts, trimmed, boost::is_any_of(","));

    // Use nlohmann json
    nlohmann::json jsonData;
    for (const auto& part : parts) {
        try {
            double value = boost::lexical_cast<double>(part);
            jsonData["values"].push_back(value);
        } catch (...) {
            jsonData["errors"].push_back(part);
        }
    }
    return jsonData.dump();
}
```

Listing 5: Source C++ code using Boost

```python
#include <boost/algorithm/string.hpp>  // Invalid in Python
#include <boost/lexical_cast.hpp>      // Invalid in Python
#include <nlohmann/json.hpp>           // Invalid in Python

def process_data(input):
    # Attempt to use C++ Boost functions in Python
    trimmed = boost.trim_copy(input)  // Invalid
    parts = []
    boost.split(parts, trimmed, boost.is_any_of(","))  // Invalid

    json_data = nlohmann.json()  // Invalid
    for part in parts:
        try:
            value = boost.lexical_cast<double>(part)  // Invalid
            json_data["values"].append(value)
        except:
            json_data["errors"].append(part)

    return json_data.dump()  // Invalid
```

Listing 6: Translated Python code retaining C++ libraries

Explanation: The model completely fails to translate the C++ Boost and nlohmann-json libraries to appropriate Python equivalents. It should have used Python's built-in string methods (`strip()`, `split()`), the `json` standard library, and potentially additional libraries like `pyphonetics` or `funcy` for more advanced operations.

A-1-4 Missing Required Libraries

Description: This error occurs when the translated code requires third-party libraries but fails to include any references to them.

Example:

```python
from lightgbm import LGBMClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np

def train_lightgbm_model(X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    model = LGBMClassifier(n_estimators=100, learning_rate=0.1)
    model.fit(X_train, y_train)

    predictions = model.predict(X_test)
    accuracy = accuracy_score(y_test, predictions)

    return model, accuracy
```

Listing 7: Source Python code using multiple ML libraries

```java
public class MLTrainer {
    public static Object[] trainLightGBMModel(double[][] X, int[] y) {
        // Missing Weka or Smile imports for train/test split
        double[][] XTrain = null;
        double[][] XTest = null;
        int[] yTrain = null;
        int[] yTest = null;

        // Missing LightGBM equivalent import
        Object model = null;  // Placeholder

        // Missing model training code
        int[] predictions = null;
        double accuracy = 0.0;  // Placeholder

        return new Object[]{model, accuracy};
    }
}
```

Listing 8: Translated Java code missing ML libraries

Explanation: The model recognizes that machine learning functionality needs translation but completely fails to identify and include appropriate Java ML libraries. It should have used `com.github.haifengl:smile-co` or `nz.ac.waikato.cms.weka:weka-stable` as equivalents for scikit-learn, and either implemented LightGBM functionality using existing Java libraries or used a different gradient boosting implementation.

A-2 API-Related Errors

A-2-1 Retaining Original PL's API

Description: This error occurs when the model keeps the source language's API call patterns.

Example:

```java
import org.jdbi.v3.core.Jdbi;
import org.jdbi.v3.core.result.ResultIterable;

public class DatabaseService {
    private final Jdbi jdbi;

    public DatabaseService(String url) {
        this.jdbi = Jdbi.create(url);
    }

    public List<String> getUsers() {
        return jdbi.withHandle(handle -> {
            ResultIterable<String> result = handle.createQuery("SELECT username FROM users")
                .mapTo(String.class);
            return result.list();
        });
    }
}
```

Listing 9: Source Java code using JDBI

```python
def get_users():
    # Attempting to use JDBI-like API in Python
    result = jdbi.withHandle(lambda handle:  # Invalid Python syntax
        handle.createQuery("SELECT username FROM users")  # Invalid
              .mapTo(str)  # Invalid
              .list()  # Invalid
    )
    return result
```

Listing 10: Translated Python retaining JDBI API patterns

Explanation: The model incorrectly tries to use JDBI's Java-specific API patterns in Python. Instead, it should use Python's database libraries like `peewee` or `sqlite3` with Pythonic patterns like context managers and cursor operations.

A-2-2 Calling a Non-Existent API

Description: This error occurs when the model invokes API methods that don't exist in the target library.

Example:

```python
import requests
from requests.exceptions import RequestException

def fetch_api_data(url, headers=None):
    try:
        response = requests.get(url, headers=headers, timeout=30)
        response.raise_for_status()
        return response.json()
    except RequestException as e:
        print(f"Request failed: {e}")
        return None
```

Listing 11: Source Python code using Requests

```java
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

public class ApiClient {
    public static Object fetchApiData(String url, Map<String, String> headers) {
        OkHttpClient client = new OkHttpClient.Builder()
            .timeout(30, TimeUnit.SECONDS)  // Correct
            .build();

        Request.Builder requestBuilder = new Request.Builder().url(url);
        if (headers != null) {
            // Non-existent method - OkHttp uses addHeader, not headers()
            requestBuilder.headers(headers);  // Invalid
        }

        try (Response response = client.newCall(requestBuilder.build()).execute()) {
            // Non-existent method - OkHttp doesn't have raise_for_status()
            response.raise_for_status();  // Invalid

            // Non-existent method - should use response.body().string() for JSON
            return response.json();  // Invalid
        } catch (IOException e) {
            System.out.println("Request failed: " + e.getMessage());
            return null;
        }
    }
}
```

Listing 12: Translated Java with non-existent OkHttp APIs

Explanation: The model invokes several non-existent methods in OkHttp: `headers()` instead of `addHeader()`, `raise_for_status()` which doesn't exist, and `response.json()` instead of parsing the response body manually with a JSON library.

A-2-3 Calling an Irrelevant API

Description: This error occurs when the model selects an API that exists but is semantically incorrect.

Example:

```cpp
#include <rapidjson/document.h>
#include <rapidjson/stringbuffer.h>
#include <rapidjson/writer.h>

std::string create_json(const std::vector<std::string>& items) {
    rapidjson::Document doc;
    doc.SetObject();
    rapidjson::Value array(rapidjson::kArrayType);

    for (const auto& item : items) {
        rapidjson::Value value;
        value.SetString(item.c_str(), doc.GetAllocator());
        array.PushBack(value, doc.GetAllocator());
    }

    doc.AddMember("items", array, doc.GetAllocator());

    rapidjson::StringBuffer buffer;
    rapidjson::Writer<rapidjson::StringBuffer> writer(buffer);
    doc.Accept(writer);

    return buffer.GetString();
}
```

Listing 13: Source C++ code using RapidJSON

```python
import json
from jsonschema import validate  # Irrelevant import

def create_json(items):
    # Using jsonschema for creation instead of validation
    schema = {  # Irrelevant schema definition
        "type": "object",
        "properties": {
            "items": {"type": "array"}
        }
    }

    data = {"items": items}

    # Irrelevant validation call for creation task
    validate(data, schema)  # Should not be here

    return json.dumps(data)  # Correct but with irrelevant steps
```

Listing 14: Translated Python using irrelevant JSON APIs

Explanation: The model incorrectly uses `jsonschema` for JSON creation when it should only be used for validation. It includes irrelevant schema definition and validation steps that are completely unnecessary for simple JSON creation, showing poor understanding of the appropriate use cases for different JSON libraries.

A-2-4 Missing an API

Description: This error occurs when the model omits necessary API calls from third-party libraries.

Example:

```python
import spacy
from spacy import displacy

def analyze_text(text):
    nlp = spacy.load("en_core_web_sm")
    doc = nlp(text)

    results = {
        "entities": [(ent.text, ent.label_) for ent in doc.ents],
        "tokens": [(token.text, token.pos_) for token in doc],
        "sentences": [sent.text for sent in doc.sents]
    }

    # Generate visualization HTML
    html = displacy.render(doc, style="dep", page=True)
    results["visualization"] = html

    return results
```

Listing 15: Source Python code using Spacy

```java
import edu.stanford.nlp.pipeline.*;
import java.util.*;

public class TextAnalyzer {
    public static Map<String, Object> analyzeText(String text) {
        // Missing pipeline setup and properties configuration
        StanfordCoreNLP pipeline = null;  // Should be properly initialized

        // Missing annotation creation and processing
        Annotation document = new Annotation(text);
        pipeline.annotate(document);  // Might fail due to missing setup

        Map<String, Object> results = new HashMap<>();
        // Missing entity extraction code
        // Missing token processing code
        // Missing sentence detection code
        // Missing visualization generation code

        return results;  // Empty results
    }
}
```

Listing 16: Translated Java missing Stanford CoreNLP APIs

Explanation: The model completely omits the necessary setup and configuration steps for Stanford CoreNLP, including pipeline properties configuration, proper annotation processing, and specific API calls for extracting entities, tokens, and sentences. It also misses the visualization generation equivalent functionality.

A-3 Parameter-Related Errors

A-3-1 Parameter Type Mismatch

Description: This error occurs when the translated code passes an argument of an incorrect data type to a function or method. This often happens when the model misunderstands a library's expected input type, which can be particularly tricky when translating between languages with different type systems (e.g., Python's dynamic typing vs. Java's static typing).

Example:

```
import pandas as pd
import numpy as np

data = {’A’: [’foo’, ’bar’, ’foo’, ’bar’],
        ’B’: [1, 2, 3, 4],
        ’C’: [2.0, 5.0, 8.0, 1.0]}
df = pd.DataFrame(data)

# Calculate the mean of column ’C’ for each group in column ’A’
grouped = df.groupby(’A’).agg({’C’: np.mean})
print(grouped)
```

Listing 17: Source Python code using ‘pandas‘ to group and aggregate data

```
import tech.tablesaw.api.Table;
import tech.tablesaw.aggregate.AggregateFunction;
import static tech.tablesaw.aggregate.AggregateFunctions.mean;

public class DataAggregator {
    public static void main(String[] args) {
        Table table = Table.create("data")
            .addColumns(
                StringColumn.create("A", new String[]{"foo", "bar", "foo", "bar"}),
                IntColumn.create("B", new int[]{1, 2, 3, 4}),
                DoubleColumn.create("C", new double[]{2.0, 5.0, 8.0, 1.0})
            );

        // Error: Parameter type mismatch. The ‘agg‘ method requires a Column and a
    function.
        // It should be ‘mean.on(table.doubleColumn("C"))‘
        Table grouped = table.by("A").agg(table.doubleColumn("C"), "mean");
        System.out.println(grouped.print());
    }
}
```

Listing 18: Translated Java code with a parameter type mismatch

Explanation: The Python code uses a dictionary to specify the aggregation function (‘np.mean‘) and the target column (‘C‘). The translated Java code, using the ‘Tablesaw‘ library, attempts to use ‘table.doubleColumn("C")‘ and the string ‘"mean"‘ as arguments to the ‘agg‘ method. However, this method's signature does not accept a ‘String‘ for the function. The correct usage would be to pass a predefined ‘AggregateFunction‘ object, such as ‘mean‘. This error demonstrates the model's failure to correctly translate the conceptual mapping of ’column’ and ’function’ into the specific, type-safe API signature of the target language library.

A-3-2 Parameter Count Errors

Description: This error occurs when the model calls an API with the wrong number of arguments. This is a clear-cut syntax error in most languages and indicates that the model has a flawed understanding of the API’s signature.

Example:

```python
import shutilwhich

# Searches for a file in a given directory list
# The second parameter is optional
path = shutilwhich.which(’myscript.sh’, [’/usr/bin’, ’/usr/local/bin’])
print(path)
```

Listing 19: Source Python code using ‘shutilwhich‘ for path resolution

```java
import org.apache.commons.io.FileUtils;
import java.io.File;

public class FileUtilsExample {
    public static void main(String[] args) {
        // Error: The ‘which‘ method from ‘commons-io‘ does not have a 2-parameter
    overload.
        // It should be ‘ExecutableFinder.find("myscript.sh", new String[]{"/usr/bin",
     "/usr/local/bin"})‘
        // or a similar method from ‘commons-exec‘.
        File path = FileUtils.which("myscript.sh", "/usr/bin", "/usr/local/bin");

        // This is a direct, incorrect translation of the parameter list.

        System.out.println(path.getAbsolutePath());
    }
}
```

Listing 20: Translated Java code with a parameter count error

Explanation: The Python source code uses ‘shutilwhich.which‘ which can take a list of directories as an optional second argument. The translated Java code incorrectly uses ‘FileUtils.which‘ (which is a non-existent method from ‘commons-io‘) and, even more critically, passes three separate string arguments to it. This demonstrates a failure to correctly map Python’s flexible argument handling to Java’s rigid method signature. The model appears to have translated the arguments one by one without considering the function’s actual parameter list, leading to a fatal compilation error.

A-3-3 Return Value Type Errors

Description: This error occurs when the translated code misinterprets the return type of a function or method. It may attempt to assign the result of a function to a variable of an incompatible type, leading to a compilation error, or to a subtle runtime bug if a type cast is performed incorrectly.

Example:

```python
import statsmodels.api as sm
import numpy as np

# A simple linear regression model
X = np.random.rand(10, 2)
y = np.random.rand(10)
model = sm.OLS(y, X).fit()
# The ‘summary()‘ method returns a string
summary_str = model.summary()
print(summary_str)
```

Listing 21: Source Python code using ‘statsmodels‘ for statistical analysis

```java
import org.apache.commons.math3.stat.regression.SimpleRegression;

public class StatsExample {
    public static void main(String[] args) {
        SimpleRegression regression = new SimpleRegression(true);
        // ... code to add data points ...

        // The ‘getRSquare()‘ method returns a double
        // The model incorrectly tries to assign it to a String variable
        String rSquared = regression.getRSquare(); // Error: Incompatible types
        System.out.println("R-squared: " + rSquared);
    }
}
```

Listing 22: Translated Java code with a return value type error

Explanation: The Python ‘statsmodels‘ library’s ‘summary()‘ method returns a formatted string. In the translated Java code, the model attempts to map this concept to the ‘getRSquare()‘ method from ‘commons-math3‘, which returns a ‘double‘ value. The model then tries to assign this ‘double‘ directly to a ‘String‘ variable. This is a type mismatch that the Java compiler will immediately flag as an error. This failure shows a lack of understanding of the target library’s return value contracts and a literal translation of a concept (‘summary‘) without regard for its type.

B Syntactic Errors

B-1 Syntax Structure Errors

Description: Accounting for 38.4% of syntactic failures, these are basic grammatical mistakes that make the code unparsable. They include issues like mismatched parentheses, incorrect indentation, and incomplete statements.

Example: Mismatched Parentheses

```
data = {'id': 123, 'name': 'Alice', 'roles': ['admin', 'user']}
user = data['name']
```

Listing 23: Source Python code with correct list and dictionary syntax

```
import java.util.HashMap;
import java.util.Arrays;
import java.util.List;
import java.util.Map;

public class DataProcessor {
    public static void processUserData() {
        Map<String, Object> data = new HashMap<>() {{
            put("id", 123);
            put("name", "Alice");
            put("roles", Arrays.asList("admin", "user"); // Error: Missing closing parenthesis
        }};
        String user = (String) data.get("name");
    }
}
```

Listing 24: Translated Java code with mismatched symbols

Explanation: The Java code is missing the closing parenthesis for the ‘Arrays.asList‘ method call, which is a clear syntax error. This illustrates a failure to maintain symbol balance during translation.

Example: Incorrect Indentation

```
if x > 5:
    print("Greater")
    print("Done")
```

Listing 25: Source Python code with correct indentation defining a code block

```
if (x > 5)
    System.out.println("Greater");
    System.out.println("Done"); // This line is always executed, a logical error
```

Listing 26: Translated Java code with incorrect block structure due to improper indentation

Explanation: In Python, indentation defines code blocks. The translated Java code lacks curly braces ‘‘, meaning only the first ‘System.out.println‘ is part of the ‘if‘ block, while the second line is always executed. This is a logical error resulting from a failure to translate the concept of code blocks correctly.

B-2 Language Feature Errors

Description: Accounting for 21.86% of errors, this category involves the incorrect translation of unique syntactic constructs. The model fails to convert a source language's specific feature into a valid, idiomatic expression in the target language.

Example: List Comprehension

```python
words = ['apple', 'banana', 'cherry']
lengths = [len(word) for word in words if len(word) > 5]
```

Listing 27: Source Python code using a list comprehension

```java
import java.util.ArrayList;
import java.util.List;

public class ListProcessor {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>(Arrays.asList("apple", "banana", "cherry"));
        List<Integer> lengths = new ArrayList<Integer>();
        for word in words: // Error: Non-existent syntax
            if len(word) > 5: // Error: Non-existent syntax
                lengths.add(len(word));
    }
}
```

Listing 28: Translated Java code with an incorrect syntax structure

Explanation: The model attempts a literal translation of Python's list comprehension, creating invalid syntax in Java (e.g., ‘for word in words:‘ and ‘len(word)‘). The correct, idiomatic translation would use a stream with ‘filter‘ and ‘map‘ or a standard ‘for‘ loop.

B-3 Object Definition Errors

Description: This is the largest category of syntactic errors, making up 39.73% of the total. It refers to mistakes in the definition of variables, functions, or classes. These errors stem from a failure to correctly apply the target language’s rules for declaring and initializing code structures.

Example: Variable Declaration

```
def my_function():
    result = 10
    return result
```

Listing 29: Source Python code with an implicit variable declaration

```
public class MyClass {
    public int my_function() {
        result = 10; // Error: ‘result‘ is not declared
        return result;
    }
}
```

Listing 30: Translated Java code with a variable declaration error

Explanation: In Python, a variable is declared the moment it’s assigned a value. In Java, however, a variable must be explicitly declared with its type (e.g., ‘int result = 10;‘) before use. The model’s failure to add this declaration results in a ‘Cannot find symbol‘ error.

Example: Function Definition

```
class MyClass:
    def my_method(self, arg1, arg2):
        print(f"{arg1} and {arg2}")
```

Listing 31: Source Python code with a correctly defined method

```
class MyClass {
    public my_method(int arg1, int arg2) { // Error: Missing return type
        System.out.println(arg1 + " and " + arg2);
    }
}
```

Listing 32: Translated Java code with an incorrect method definition

Explanation: The Python method definition is translated into a Java method, but the model omits the mandatory return type (‘void‘, ‘int‘, etc.). This is a basic syntax error in Java and demonstrates a misunderstanding of the target language’s function signature requirements.

C Code Generation Errors

Description: Code generation errors refer to failures where the model produces an output that is not a valid, complete, or functional code translation. This category includes generating non-code content, producing incomplete or contradictory code, and omitting essential logical blocks.

C-1 Generation of Non-Code Content

Description: This error occurs when the model includes human-readable, explanatory text or comments that are not part of the source code. This indicates a failure to strictly adhere to a code-only generation task.

Example: Indicating Inability to Complete

```python
import cvxpy as cp
import numpy as np
problem = cp.Problem(...)
problem.solve()
```

Listing 33: Source Python code using a complex library

```
I am sorry, but I cannot directly translate this complex convex optimization problem. There is no direct, one-to-one library equivalent in Java.
```

Listing 34: Translated Java output indicating a failure to translate

Explanation: Instead of attempting a translation, the model explicitly states it cannot perform the task due to the complexity and lack of an equivalent library in the target language. This is a complete failure to fulfill the user's request.

C-2 Incomplete Code Generation

Description: This error occurs when the model fails to produce a full, logically complete translation. It might stop generating code prematurely or omit crucial code blocks.

Example: Missing Critical Code Blocks

```python
def process_data_and_log(data):
    processed = []
    for item in data:
        if item > 0:
            processed.append(item * 2)
    for p in processed:
        print(f"Processed item: {p}")
    return processed
```

Listing 35: Source Python code with a crucial loop

```java
import java.util.ArrayList;
import java.util.List;

public class DataProcessor {
    public static List<Integer> processDataAndLog(List<Integer> data) {
        List<Integer> processed = new ArrayList<>();
        for (int item : data) {
            if (item > 0) {
                processed.add(item * 2);
            }
        }
        // Error: The logging loop is completely missing
        return processed;
    }
}
```

Listing 36: Translated Java code missing a critical logging block

Explanation: The source code has two main parts: data processing and logging. The translated code only includes the data processing part and completely omits the logging loop. While the code is syntactically valid, it is functionally incomplete.

C-3 Contradictory Code Generation

Description: The translated code is syntactically correct but implements a logic that directly contradicts the source code's intent or introduces inconsistencies.

Example: Inconsistent Variable Naming

```
def process_user_input(user_id):
    user_data = get_user_data(user_id)
    return user_data
```

Listing 37: Source Python code with consistent variable naming

```
public class UserManager {
    public static User processUserInput(String userId) {
        UserData userData = getUserData(userId); // Valid
        return userdata; // Error: Inconsistent naming
    }
}
```

Listing 38: Translated Java code with inconsistent variable naming

Explanation: The model uses ‘userData‘ in the declaration but then refers to it as ‘userdata‘ in the ‘return‘ statement. Since Java is case-sensitive, this leads to a ‘Cannot find symbol‘ error.

Example: Logical Contradictions

```
if temperature > 30:
    print("It's hot.")
else:
    print("It's not hot.")
```

Listing 39: Source Python code with a correct conditional check

```
public class WeatherCheck {
    public static void checkTemperature(int temperature) {
        if (temperature < 30) { // Error: Logical contradiction
            System.out.println("It's hot.");
        } else {
            System.out.println("It's not hot.");
        }
    }
}
```

Listing 40: Translated Java code with a logical contradiction

Explanation: The translated code reverses the logical condition from ‘>‘ to ‘<‘. While the code will compile and run, it produces the wrong output, demonstrating a failure to translate the core logical intent.

D Runtime Errors

Description: Runtime errors are problems that do not prevent the code from compiling but cause it to fail during execution. These are often difficult to detect without running the program.

Example: Null Pointer Dereferencing

```python
def get_name(user_data):
    if user_data is not None:
        return user_data.get(’name’)
    return "Guest"
```

Listing 41: Source Python code with a null-safe check

```java
import java.util.Map;

public class UserDataHandler {
    public static String getName(Map<String, Object> userData) {
        return (String) userData.get("name"); // NullPointerException if userData is null
    }
}
```

Listing 42: Translated Java code with a null pointer dereferencing error

Explanation: The model omits the null check from the original Python code. If the Java method is called with a ‘null‘ argument, it will throw a ‘NullPointerException‘ at runtime.

Example: Array Out-of-Bounds

```python
my_list = [10, 20, 30]
for i in range(len(my_list)):
    print(my_list[i])
```

Listing 43: Source Python code with a safe loop range

```java
public class ListProcessor {
    public static void processList() {
        int[] myList = {10, 20, 30};
        for (int i = 0; i <= myList.length; i++) { // Error: Off-by-one loop condition
            System.out.println(myList[i]); // ArrayIndexOutOfBoundsException on last iteration
        }
    }
}
```

Listing 44: Translated Java code with an array out-of-bounds error

Explanation: The translated code’s loop condition ‘i <= myList.length‘ causes it to access an index beyond the array’s bounds on the last iteration, leading to an ‘ArrayIndexOutOfBoundsException‘ at runtime.

E Other Errors

Description: This category encompasses various error patterns that evaluators found difficult to classify into the existing taxonomy. These cases represent unique or ambiguous failure modes that do not clearly align with the predefined error categories, reflecting the complex and sometimes unpredictable nature of library-related code translation challenges.

Note: For the sake of conciseness and focus, this section does not provide specific examples of these unclassified errors. Instead, we acknowledge their existence as part of the comprehensive error landscape while maintaining our primary focus on the well-defined, reproducible failure patterns that constitute the core of our analysis.