Variant analysis

Below is a comprehensive Java example that implements the logic described earlier, including:

* **Input Handling**: Takes a list of raw variants as input.
* **Loop Normalization**: Collapses consecutive identical tasks into single occurrences.
* **Automatic Parallelism Detection**:
  + Builds ordering relations between tasks across all variants.
  + Identifies unordered sets of tasks as potential parallel sets using a heuristic approach.
* **Canonicalizing Parallel Blocks**: Replaces contiguous segments matching these parallel sets with a canonical representation (sorted list).
* **Subset Detection**: Removes variants that are prefixes of more complete variants.
* **Extendability**: The code is organized so you can add more conditions that cause variant explosion if needed.

**Note**:  
This code uses a heuristic for parallel detection. In real-world scenarios, you would likely use established process mining algorithms or libraries. The heuristic might not perfectly capture all forms of parallelism, but it demonstrates the approach conceptually.

**Java Code (Single File Example)**:

import java.util.\*;

import java.util.stream.Collectors;

public class ProcessVariantNormalizer {

// A simple Pair class for handling relations between tasks

// We assume tasks are strings

static class Pair {

String first;

String second;

Pair(String f, String s) {

this.first = f;

this.second = s;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (!(o instanceof Pair)) return false;

Pair pair = (Pair) o;

return Objects.equals(first, pair.first) && Objects.equals(second, pair.second);

}

@Override

public int hashCode() {

return Objects.hash(first, second);

}

}

/\*\*

\* Collapse consecutive identical tasks into one occurrence.

\* E.g., [A, B, B, B, C] -> [A, B, C]

\*/

public static List<String> normalizeLoops(List<String> variant) {

if (variant.isEmpty()) return variant;

List<String> normalized = new ArrayList<>();

normalized.add(variant.get(0));

for (int i = 1; i < variant.size(); i++) {

String task = variant.get(i);

if (!task.equals(normalized.get(normalized.size() - 1))) {

normalized.add(task);

}

}

return normalized;

}

/\*\*

\* Check if candidate is a prefix of any existing more complete variant.

\*/

public static boolean isSubsetVariant(List<Object> candidate, List<List<Object>> existing) {

for (List<Object> ev : existing) {

if (candidate.size() < ev.size()) {

boolean prefix = true;

for (int i = 0; i < candidate.size(); i++) {

if (!candidate.get(i).equals(ev.get(i))) {

prefix = false;

break;

}

}

if (prefix) return true;

}

}

return false;

}

/\*\*

\* Build ordering relations between tasks.

\* For each pair (X,Y), count how many times X appears before Y across all variants.

\*/

public static Tuple buildTaskRelations(List<List<String>> variants) {

Set<String> tasks = new HashSet<>();

for (List<String> v : variants) {

tasks.addAll(v);

}

// Relations: map of (X,Y) -> count of X\_before\_Y

Map<Pair, Integer> relations = new HashMap<>();

// For each variant, record order relations

for (List<String> v : variants) {

Set<String> seen = new HashSet<>();

for (String current : v) {

for (String s : seen) {

Pair p = new Pair(s, current);

relations.put(p, relations.getOrDefault(p, 0) + 1);

}

seen.add(current);

}

}

return new Tuple(tasks, relations);

}

static class Tuple {

Set<String> tasks;

Map<Pair, Integer> relations;

Tuple(Set<String> tasks, Map<Pair, Integer> relations) {

this.tasks = tasks;

this.relations = relations;

}

}

/\*\*

\* Identify parallel sets of tasks based on ordering relations.

\* Heuristic:

\* - For tasks X,Y, count X\_before\_Y and Y\_before\_X.

\* - If no strong ordering (ratio < threshold), they are considered unordered (parallel candidates).

\* - Then find connected components of unordered tasks to form parallel sets.

\*/

public static List<Set<String>> identifyParallelSets(Set<String> tasks, Map<Pair, Integer> relations, double thresholdRatio) {

List<String> taskList = new ArrayList<>(tasks);

Collections.sort(taskList);

// Build symmetrical counts

// For each pair (X,Y), get X\_before\_Y and Y\_before\_X

Map<Pair, int[]> counts = new HashMap<>();

for (String x : taskList) {

for (String y : taskList) {

if (!x.equals(y)) {

int x\_before\_y = relations.getOrDefault(new Pair(x, y), 0);

int y\_before\_x = relations.getOrDefault(new Pair(y, x), 0);

counts.put(new Pair(x,y), new int[]{x\_before\_y, y\_before\_x});

}

}

}

// Determine unordered pairs

Set<Pair> unorderedPairs = new HashSet<>();

// Only check each pair once (e.g. x<y to avoid duplicates)

for (String x : taskList) {

for (String y : taskList) {

if (x.compareTo(y) < 0) {

int[] vals = counts.get(new Pair(x,y));

if (vals != null) {

int xy = vals[0];

int yx = vals[1];

int total = xy + yx;

if (total > 0) {

double ratio = Math.max(xy, yx) / (double) total;

if (ratio < thresholdRatio) {

// unordered

unorderedPairs.add(new Pair(x,y));

}

}

}

}

}

}

// Build graph of unordered relationships

Map<String, Set<String>> graph = new HashMap<>();

for (String t : taskList) {

graph.put(t, new HashSet<>());

}

for (Pair p : unorderedPairs) {

graph.get(p.first).add(p.second);

graph.get(p.second).add(p.first);

}

// Find connected components

Set<String> visited = new HashSet<>();

List<Set<String>> parallelSets = new ArrayList<>();

for (String t : taskList) {

if (!visited.contains(t)) {

Set<String> comp = dfs(graph, t, visited);

parallelSets.add(comp);

}

}

return parallelSets;

}

private static Set<String> dfs(Map<String, Set<String>> graph, String start, Set<String> visited) {

Stack<String> stack = new Stack<>();

stack.push(start);

Set<String> comp = new HashSet<>();

while (!stack.isEmpty()) {

String node = stack.pop();

if (!visited.contains(node)) {

visited.add(node);

comp.add(node);

for (String nei : graph.get(node)) {

if (!visited.contains(nei)) {

stack.push(nei);

}

}

}

}

return comp;

}

/\*\*

\* Canonicalize parallel blocks:

\* Attempt to replace contiguous segments that match a parallel set with a sorted block.

\* We match largest sets first to avoid partial matches overshadowing bigger sets.

\*/

public static List<Object> canonicalizeParallelBlocks(List<String> variant, List<Set<String>> parallelSets) {

// Sort parallel sets by size descending

List<Set<String>> sortedParallelSets = new ArrayList<>(parallelSets);

sortedParallelSets.sort((a,b)->Integer.compare(b.size(), a.size()));

List<Object> result = new ArrayList<>(variant);

boolean changed = true;

while (changed) {

changed = false;

for (Set<String> pset : sortedParallelSets) {

int p\_len = pset.size();

// Skip singletons because they won't form a block

if (p\_len <= 1) {

continue;

}

for (int i = 0; i <= result.size() - p\_len; i++) {

// Extract a window

List<Object> window = new ArrayList<>(result.subList(i, i+p\_len));

// Check if all are strings and match pset

if (allMatchSet(window, pset)) {

// Replace this segment

List<String> block = new ArrayList<>(pset);

Collections.sort(block);

List<Object> newResult = new ArrayList<>(result.subList(0, i));

newResult.add(block);

newResult.addAll(result.subList(i+p\_len, result.size()));

result = newResult;

changed = true;

break;

}

}

if (changed) break;

}

}

return result;

}

private static boolean allMatchSet(List<Object> window, Set<String> pset) {

if (window.size() != pset.size()) return false;

Set<String> wset = new HashSet<>();

for (Object o : window) {

if (!(o instanceof String)) return false;

wset.add((String)o);

}

return wset.equals(pset);

}

/\*\*

\* Full normalization pipeline:

\* 1. Normalize loops.

\* 2. Detect parallel sets automatically.

\* 3. Canonicalize parallel blocks.

\* 4. Remove subset variants.

\*/

public static List<List<Object>> normalizeVariants(List<List<String>> rawVariants) {

// Step 1: Normalize loops

List<List<String>> loopNormalized = rawVariants.stream()

.map(ProcessVariantNormalizer::normalizeLoops)

.collect(Collectors.toList());

// Step 2: Detect parallel sets automatically

Tuple t = buildTaskRelations(loopNormalized);

// A threshold ratio. If no direction strongly dominates, consider them parallel.

double thresholdRatio = 0.7;

List<Set<String>> parallelSets = identifyParallelSets(t.tasks, t.relations, thresholdRatio);

// Step 3 & 4: Canonicalize and remove subsets

List<List<Object>> finalVariants = new ArrayList<>();

for (List<String> rv : loopNormalized) {

List<Object> canonical = canonicalizeParallelBlocks(rv, parallelSets);

if (!isSubsetVariant(canonical, finalVariants)) {

finalVariants.add(canonical);

}

}

return finalVariants;

}

public static void main(String[] args) {

// Example input

List<List<String>> rawVariants = new ArrayList<>();

rawVariants.add(Arrays.asList("A","B","B","B","C","D","E")); // Loops of B

rawVariants.add(Arrays.asList("A","B","C","D","E")); // Clean variant

rawVariants.add(Arrays.asList("A","C","B","D","E")); // Different order suggests parallel

rawVariants.add(Arrays.asList("A","B","C","E","D")); // Another permutation

rawVariants.add(Arrays.asList("A","B")); // Incomplete (subset)

rawVariants.add(Arrays.asList("A","B","C")); // Another incomplete subset

rawVariants.add(Arrays.asList("A","D","C","B","E")); // Another permutation

rawVariants.add(Arrays.asList("X","Y","Z")); // Different sequence

rawVariants.add(Arrays.asList("X","Z","Y")); // Permutation of X,Y,Z

rawVariants.add(Arrays.asList("X","Y","Z","Z","Z")); // Loops in a different sequence

List<List<Object>> mainVariants = normalizeVariants(rawVariants);

System.out.println("Final normalized main variants:");

for (List<Object> v : mainVariants) {

System.out.println(v);

}

}

}

**What this Java code does**:

* **normalizeLoops**: Removes consecutive duplicates.
* **buildTaskRelations**: For every pair of tasks, counts how often one appears before the other across all variants.
* **identifyParallelSets**: Uses the relations to find tasks that are unordered and groups them into parallel sets.
* **canonicalizeParallelBlocks**: Replaces any contiguous segments that form a parallel set with a canonical sorted block.
* **isSubsetVariant**: Ensures we do not include partial variants that are prefixes of more complete variants.
* **normalizeVariants**: Ties all steps together into a single pipeline.

**Result**: The code prints out a reduced set of main variants after applying normalization, parallel detection, and subset removal. You can adjust parameters, thresholds, or incorporate more complex logic as needed.

Below is a comprehensive Python example that demonstrates:

1. **Input Handling**: Provide a set of raw variants as input.
2. **Automatic Parallelism Detection**: Identify parallel (concurrent) sets of tasks automatically by analyzing the ordering of tasks across all variants.
3. **Normalization Steps**:
   * **Loop Normalization**: Collapse consecutive identical tasks in each variant.
   * **Parallel Block Canonicalization**: Using the discovered parallel sets, replace unordered parallel segments with a canonical representation.
   * **Subset Variant Detection**: Remove variants that are prefixes (subsets) of more complete variants.
4. **Other Variant Explosion Conditions**: The code is structured so that adding more rules or conditions (like noise filtering, optional task handling, etc.) is straightforward.

**Heuristic for Parallel Detection** (Simple Example):

* We compare each pair of distinct tasks and count how many times one appears before the other across all variants.
* If for tasks X and Y, the counts X\_before\_Y and Y\_before\_X are roughly equal (no strong dominance), we treat them as potentially parallel.
* We then group such mutually unordered tasks into sets. For simplicity, this example tries to find sets of tasks that are mutually unordered. In reality, a more sophisticated clustering or graph-based algorithm might be used.

**Note:** This heuristic is simplified and may not perfectly capture complex parallelism scenarios. For advanced concurrency detection, consider using established process mining algorithms or libraries (e.g., PM4Py).

**Complete Python Code Example**

from typing import List, Set, Tuple

from collections import defaultdict

def normalize\_loops(variant: List[str]) -> List[str]:

"""Collapse consecutive identical tasks into one occurrence."""

if not variant:

return variant

normalized = [variant[0]]

for task in variant[1:]:

if task != normalized[-1]:

normalized.append(task)

return normalized

def is\_subset\_variant(candidate: List[object], existing: List[List[object]]) -> bool:

"""Check if candidate is a prefix of any existing more complete variant."""

for ev in existing:

if len(candidate) < len(ev):

if all(c == e for c, e in zip(candidate, ev)):

return True

return False

def build\_task\_relations(variants: List[List[str]]) -> Tuple[Set[str], dict]:

"""

Analyze all variants to build ordering relations between tasks.

Returns:

tasks: Set of all distinct tasks

relations: A dictionary keyed by (X,Y) with counts of how often X occurs before Y.

"""

tasks = set()

for v in variants:

tasks.update(v)

tasks = set(tasks)

# Count how many times X appears before Y

relations = defaultdict(int)

# For each variant, note the ordering of tasks

# If v = [A, B, C], then A before B, A before C, B before C

for v in variants:

seen = set()

for i in range(len(v)):

for s in seen:

# s appears before v[i]

relations[(s, v[i])] += 1

seen.add(v[i])

return tasks, relations

def identify\_parallel\_sets(tasks: Set[str], relations: dict, threshold\_ratio: float = 0.7) -> List[Set[str]]:

"""

Identify parallel sets of tasks based on the ordering relations.

Heuristic:

- For each pair (X,Y), we have counts X\_before\_Y and Y\_before\_X.

- Compute a ratio = max(X\_before\_Y, Y\_before\_X) / (X\_before\_Y + Y\_before\_X).

- If ratio < threshold\_ratio, it means no strong ordering (i.e., they appear in random order).

- Those tasks might be parallel candidates.

Then group mutually unordered tasks into sets. A very simplified approach:

- Start each task in its own set.

- Merge sets if tasks across sets are mutually unordered.

Note: This is a simplistic heuristic and might over-group or under-group.

"""

# Build a symmetrical structure of dominance

# For each pair (X,Y), find X\_before\_Y and Y\_before\_X

# If ratio < threshold\_ratio, consider them unordered (parallel candidates)

task\_list = sorted(tasks)

# Build counts for both directions

counts = defaultdict(lambda: (0,0))

# We already have relations like (X,Y): count. Get the opposite direction too.

# relations[(X,Y)] = X\_before\_Y count

# If (Y,X) not present, it's zero.

# We'll just ensure we have a symmetrical structure

for x in task\_list:

for y in task\_list:

if x != y:

x\_before\_y = relations.get((x, y), 0)

y\_before\_x = relations.get((y, x), 0)

counts[(x,y)] = (x\_before\_y, y\_before\_x)

# Determine unordered relationships

unordered\_pairs = set()

for (x, y), (xy\_count, yx\_count) in counts.items():

if x < y: # Check each pair only once

total = xy\_count + yx\_count

if total > 0:

# If neither direction dominates strongly

ratio = max(xy\_count, yx\_count) / total

if ratio < threshold\_ratio:

# x and y are unordered

unordered\_pairs.add((x, y))

# Group tasks by unordered relations

# We'll use a simple union-find or connected components approach:

# Build a graph where edges represent unordered relations.

graph = {t: set() for t in task\_list}

for (x, y) in unordered\_pairs:

graph[x].add(y)

graph[y].add(x)

# Find connected components in this graph

visited = set()

parallel\_sets = []

def dfs(start):

stack = [start]

comp = set()

while stack:

node = stack.pop()

if node not in visited:

visited.add(node)

comp.add(node)

for nei in graph[node]:

if nei not in visited:

stack.append(nei)

return comp

for t in task\_list:

if t not in visited:

comp = dfs(t)

# If a component has more than one task, it's considered a parallel set.

# If it's a single task with no strong ordering to others, we leave it as is.

# However, a single task can also be considered a trivial "parallel set."

parallel\_sets.append(comp)

# Optionally filter out singletons if desired. Here we keep them all,

# meaning single tasks are their own "parallel block" if no ordering found.

# If you only want sets that are definitely parallel (more than one task),

# you could filter out sets of size 1 that don't show parallelism.

# Filter: Keep only sets that have at least some evidence of unordered tasks.

# Actually, let's keep all. Parallel sets of size 1 won't do harm.

return parallel\_sets

def canonicalize\_parallel\_blocks(variant: List[str], parallel\_sets: List[Set[str]]) -> List[object]:

"""

Replace segments of variants that form a parallel set with a canonical form.

This is tricky because we need to find contiguous segments that match a parallel set.

A simple approach:

- Sort parallel sets by size (larger sets first) to match bigger sets first.

- For each parallel set, try to find a contiguous window in the variant that matches it.

- Replace that window with a sorted list (canonical form).

Note: This could break down if sets overlap or tasks appear multiple times.

Heuristics may vary. For simplicity, we greedily match largest sets first.

"""

# Sort parallel sets by size descending

parallel\_sets\_sorted = sorted(parallel\_sets, key=len, reverse=True)

result = variant[:]

changed = True

while changed:

changed = False

for pset in parallel\_sets\_sorted:

p\_len = len(pset)

if p\_len <= 1:

# A single-task "parallel" set doesn't provide a transformation.

continue

i = 0

while i <= len(result) - p\_len:

window = result[i:i+p\_len]

if set(window) == pset:

# Replace with a sorted list representing the parallel block

block = list(sorted(pset))

result = result[:i] + [block] + result[i+p\_len:]

changed = True

break # restart from the beginning after one replacement

if changed:

break

return result

def normalize\_variants(raw\_variants: List[List[str]]) -> List[List[object]]:

"""

Full normalization pipeline:

1. Detect parallel sets automatically.

2. Normalize loops.

3. Canonicalize parallel blocks.

4. Remove subset variants.

"""

# Normalize loops first to reduce noise in parallel detection

loop\_normalized = [normalize\_loops(rv) for rv in raw\_variants]

# Detect parallel sets automatically

tasks, relations = build\_task\_relations(loop\_normalized)

parallel\_sets = identify\_parallel\_sets(tasks, relations)

# Now canonicalize and remove subsets

final\_variants = []

for rv in loop\_normalized:

canonical = canonicalize\_parallel\_blocks(rv, parallel\_sets)

if not is\_subset\_variant(canonical, final\_variants):

final\_variants.append(canonical)

return final\_variants

if \_\_name\_\_ == "\_\_main\_\_":

# Example input: A set of variants that may have loops, parallel behavior, subsets, etc.

raw\_variants = [

['A','B','B','B','C','D','E'], # Loops of B

['A','B','C','D','E'], # Clean variant

['A','C','B','D','E'], # Different order, suggests B and C might be parallel

['A','B','C','E','D'], # Another permutation of C,D,E region

['A','B'], # Incomplete (subset)

['A','B','C'], # Another incomplete subset

['A','D','C','B','E'], # Yet another permutation

['X','Y','Z'], # Different sequence altogether to show separate structure

['X','Z','Y'], # Permutation of X,Y,Z

['X','Y','Z','Z','Z'] # Loops in a different sequence

]

# Normalize and reduce variants

main\_variants = normalize\_variants(raw\_variants)

print("Final normalized main variants:")

for v in main\_variants:

print(v)

**Explanation**

* **Loop Normalization**: First we normalize loops to ensure unnecessary repeated sequences do not skew the parallel detection.
* **Parallel Detection**: The identify\_parallel\_sets function uses a heuristic that looks at the ordering relations of tasks across all variants. Tasks that do not show a consistent ordering are grouped together as parallel sets.
* **Canonicalization**: Once parallel sets are identified, canonicalize\_parallel\_blocks attempts to replace contiguous segments of a variant that form a parallel set with a canonical sorted list. This step reduces all permutations of parallel tasks to a single variant form.
* **Subset Removal**: Finally, if one variant is a prefix of another (incomplete trace), it is removed from the main set to avoid counting partial traces as distinct variants.

This end-to-end example thus takes raw variants and outputs a reduced, more semantically meaningful set of main variants.

When analyzing real-world event logs, it often becomes evident that the sheer number of unique execution sequences—or “variants”—is far greater than anticipated. Understanding the root causes behind this variant explosion is crucial, as it allows us to devise strategies to reduce, consolidate, or filter out unnecessary complexity. Below are a variety of scenarios and factors that can lead to an excessive number of variants in a process:

**1. Parallelism and Concurrency**

**Scenario**: Multiple tasks can be executed in parallel rather than strictly sequentially.  
**Effect**: Parallel execution paths multiply the number of possible sequences, especially if tasks can start or finish in different orders. Even if the logical structure is just two tasks in parallel, the log may record them in various partial orders. For example:

* If Task A and Task B can both run concurrently, you might see variants like [A → B] in one case and [B → A] in another. This doubling effect scales quickly with more parallel tasks, resulting in a combinatorial explosion of variants.

**2. Optional Steps and Conditional Behavior**

**Scenario**: Certain tasks are conditionally executed or optional.  
**Effect**: Whenever a process contains steps that may or may not happen depending on specific conditions (data values, user decisions, resource availability), you get at least two classes of variants: those that include the optional step and those that skip it.  
**Example**: An “Approval” step may occur only if the invoice amount is above a certain threshold. Over many cases, conditions yield many unique traces with or without these optional tasks.

**3. Incomplete Event Traces**

**Scenario**: Partial logging, missing events, or cases that are never completed.  
**Effect**: If some events are not recorded, are lost due to system errors, or if a process instance is not yet finished at the time of log extraction, the sequence may appear truncated. This partial or incomplete trace might differ structurally from completed ones, hence counting as a separate variant.  
**Example**: A case that ends prematurely due to a technical issue will look different from one that followed the full “happy path,” thus increasing the variant count.

**4. Data Quality Issues and Noise**

**Scenario**: Logging mistakes, inconsistent event labeling, timestamp errors, or duplications.  
**Effect**: Data quality problems often introduce artificial variability. Duplicate events that shouldn't be there, mislabeled tasks, or out-of-order timestamps can create unique, spurious variants that do not correspond to actual process differences.  
**Example**: An activity recorded twice by error (“Review” → “Review” again) due to a logging glitch creates an unnecessary variant.

**5. Overly Fine-Grained Activity Definitions**

**Scenario**: Very granular activity labels that capture too many small steps rather than consolidated business tasks.  
**Effect**: A single conceptual business activity split into multiple sub-steps (e.g., “Check Inventory Step 1,” “Check Inventory Step 2,” “Check Inventory Final”) can dramatically increase the number of variants. Slight differences in the execution order or the omission of a sub-step will create unique traces.  
**Example**: If a user action is broken into three or four micro-steps, their different combinations or partial executions quickly inflate the variant count.

**6. Rework and Iterations**

**Scenario**: Tasks may be repeated multiple times due to errors, quality checks, or iterations in the workflow.  
**Effect**: Repeated loops of certain activities can yield variants where the same tasks appear in different frequencies or orders. One case might perform a rework cycle twice, another might do it four times, creating a multitude of distinct sequences.  
**Example**: A task “Review Document” done once in one case and done three times in another, purely because the reviewer wasn’t satisfied initially, results in different variants.

**7. Multiple Resource Involvement**

**Scenario**: The same task could be performed by different roles, departments, or even automated vs. manual resources.  
**Effect**: If the event log records the resource as part of the activity label (e.g., “Approve (Manager A)” vs. “Approve (Manager B)”), variations in who performed the task become variations in the trace.  
**Example**: The same approval action done by different approvers results in distinct variants, even though logically it’s the same step.

**8. Rare or Exceptional Cases**

**Scenario**: Some cases may follow unusual paths due to special exceptions, error handling procedures, or one-off customer requests.  
**Effect**: Even if these exceptional paths occur infrequently, each creates a unique variant. Over many cases, rare exceptions accumulate into a large number of unique patterns.  
**Example**: A special discount step executed only for a handful of VIP customers still creates unique trace patterns.

**9. External Interruptions and Wait States**

**Scenario**: Processes may be paused, waiting for external input, system availability, or regulatory approval.  
**Effect**: Different timing patterns or insertion of waiting activities can show up as different sequences in the log. If the waiting or pause is recorded as an event or leads to subtle reorderings of subsequent steps, you get more variants.  
**Example**: A “Wait for Customer Reply” event may sometimes appear after a preliminary check, other times after a partial review, leading to trace variability.

**10. Timing/Ordering Variability Without Logical Difference**

**Scenario**: Tasks that can occur in either order without changing the underlying logic (due to truly parallel capabilities or because the order doesn’t matter).  
**Effect**: Even if logically equivalent, the fact that logs record them in different sequences leads to distinct variants.  
**Example**: Two equally valid processing sequences differ only in the order of completion of certain interchangeable steps.

**11. Logging at Different Granularities or Phases**

**Scenario**: Some logs might capture system-level events (like database commits), others capture high-level business steps.  
**Effect**: Differences in the level of abstraction cause multiple variants—e.g., one system logs every system call (leading to many micro-events), while another only logs key business milestones. Mixing these granularities increases variant diversity.

**12. Inconsistent Application of Business Rules or Policies**

**Scenario**: If business rules are not uniformly applied or if different subunits follow slightly varied procedures, the resulting event logs differ.  
**Effect**: The process itself might be nominally the same, but local differences in interpretation cause variant proliferation.  
**Example**: One team always performs a QA check before approval; another team does it only after an approval. Both are valid, but each yields a distinct trace pattern.

**Summary**

In essence, anything that introduces additional complexity, flexibility, or inconsistency in how a process is executed or recorded can contribute to variant proliferation. The main causes can be grouped into several categories:

* **Execution Flexibility**: Parallel paths, optional tasks, rework loops, and variable resource assignments.
* **Data and Logging Issues**: Missing events, noise, inconsistent labeling, and varying levels of granularity.
* **Rare and Exceptional Cases**: Special handling steps, unique conditions, and external interruptions.

By identifying these causes, organizations can take steps to either standardize their processes, improve their logging practices (consolidate activity labels, filter noise, handle missing data), or employ more sophisticated mining techniques that cluster, abstract, or prune out low-frequency variants. Once the root causes of variant proliferation are known, it becomes possible to mitigate them and produce more manageable, interpretable process models.

import java.util.\*;

import java.util.stream.Collectors;

public class ProcessVariantNormalizer {

/\*\*

\* Normalize loops by collapsing consecutive identical tasks.

\* E.g. A,B,B,B,C -> A,B,C

\*/

public static List<String> normalizeLoops(List<String> variant) {

if (variant.isEmpty()) return variant;

List<String> normalized = new ArrayList<>();

normalized.add(variant.get(0));

for (int i = 1; i < variant.size(); i++) {

String task = variant.get(i);

if (!task.equals(normalized.get(normalized.size() - 1))) {

normalized.add(task);

}

}

return normalized;

}

/\*\*

\* Canonicalize parallel blocks.

\* Given known parallel sets, if we detect that a contiguous segment

\* matches a parallel set, we replace it with a sorted representation.

\*

\* For example:

\* variant: A, B, C, D, E

\* parallelSets: [ {B,C,D} ]

\* will turn into: A, (B,C,D), E where (B,C,D) is a canonical form.

\*/

public static List<Object> canonicalizeParallelBlocks(List<String> variant, List<Set<String>> parallelSets) {

List<Object> result = new ArrayList<>();

int i = 0;

while (i < variant.size()) {

boolean matched = false;

for (Set<String> pset : parallelSets) {

int pLen = pset.size();

if (i + pLen <= variant.size()) {

List<String> window = variant.subList(i, i + pLen);

Set<String> windowSet = new HashSet<>(window);

if (windowSet.equals(pset)) {

// Found a parallel block

List<String> sortedBlock = new ArrayList<>(windowSet);

Collections.sort(sortedBlock);

result.add(sortedBlock);

i += pLen;

matched = true;

break;

}

}

}

if (!matched) {

// No parallel set matched here

result.add(variant.get(i));

i++;

}

}

return result;

}

/\*\*

\* Check if a variant is a prefix (subset) of any existing more complete variant.

\* If so, we do not treat it as a main variant.

\*/

public static boolean isSubsetVariant(List<Object> candidate, List<List<Object>> existing) {

// Check if 'candidate' is a prefix of any variant in 'existing'

for (List<Object> ev : existing) {

if (candidate.size() < ev.size()) {

boolean isPrefix = true;

for (int i = 0; i < candidate.size(); i++) {

if (!candidate.get(i).equals(ev.get(i))) {

isPrefix = false;

break;

}

}

if (isPrefix) return true;

}

}

return false;

}

/\*\*

\* Main normalization method:

\* 1. Normalizes loops

\* 2. Canonicalizes parallel blocks

\* 3. Removes subset variants

\*/

public static List<List<Object>> normalizeVariants(List<List<String>> rawVariants, List<Set<String>> parallelSets) {

List<List<Object>> normalizedVariants = new ArrayList<>();

for (List<String> rv : rawVariants) {

// Step 1: Normalize loops

List<String> nl = normalizeLoops(rv);

// Step 2: Canonicalize parallel blocks

List<Object> nc = canonicalizeParallelBlocks(nl, parallelSets);

// Step 3: Check subset

if (!isSubsetVariant(nc, normalizedVariants)) {

// If not a subset, add to final list

normalizedVariants.add(nc);

}

}

return normalizedVariants;

}

public static void main(String[] args) {

// Example input

List<List<String>> rawVariants = Arrays.asList(

Arrays.asList("A","B","B","B","C","D","E"), // Loops

Arrays.asList("A","B","C","D","E"), // Clean

Arrays.asList("A","C","B","D","E"), // Parallel permutation

Arrays.asList("A","B","C","E","D"), // Another parallel permutation

Arrays.asList("A","B"), // Incomplete

Arrays.asList("A","B","C"), // Another incomplete

Arrays.asList("A","D","C","B","E") // Another permutation

);

// Known parallel sets

List<Set<String>> parallelSets = Collections.singletonList(new HashSet<>(Arrays.asList("B","C","D")));

List<List<Object>> mainVariants = normalizeVariants(rawVariants, parallelSets);

System.out.println("Final normalized variants:");

for (List<Object> v : mainVariants) {

System.out.println(v);

}

}

}