The "Semi-ATE/STDF" library is designed for parsing and working with STDF files in Python, and it aims to convert STDF data into pandas DataFrames for easier analysis. While the library itself is excellent for reading and manipulating STDF data, it doesn't have a direct, built-in function like merge\_and\_keep\_final\_results that handles the specific logic of merging multiple STDF files from the same lot, where later files represent re-tests and you only want the *final* result for each part/test.

This kind of merging requires custom logic because the STDF format is a stream of records, and identifying "redundant" tests and "final" results for a specific part/test combination requires understanding the test flow and potentially tracking part IDs and test numbers across multiple files.

Here's a conceptual outline and a Python code example using Semi-ATE/STDF and pandas to achieve your goal. This approach assumes:

1. Unique Part Identification: Each part tested has a unique identifier (e.g., in the PRR record).
2. Test Identification: Each test within a part's sequence can be uniquely identified (e.g., by test number, test name, or a combination).
3. Retest Logic: A later test for the same part/test combination in a subsequent file supersedes previous results. The order of files matters: the last file processed for a given part/test combination is considered the "final" result.

Core Idea:

1. Parse each STDF file: Convert the relevant records (like Part Results (PRR), Test Results (PTR), Site Results (STR), etc.) into pandas DataFrames.
2. Identify relevant fields: Determine which columns in your DataFrames uniquely identify a part (e.g., HEAD\_NUM, SITE\_NUM, PART\_ID) and a specific test within that part (e.g., TEST\_NUM, TEST\_NAM).
3. Merge DataFrames: Combine the DataFrames from all STDF files.
4. Deduplicate and keep final results: Use a combination of sorting and drop\_duplicates to keep only the latest (final) result for each part-test combination.

Example Python Code:

First, ensure you have the Semi-ATE and pandas libraries installed:

Bash

pip install Semi-ATE pandas

Now, here's the Python code:

Python

import os

import pandas as pd

from Semi\_ATE.STDF import V4

def parse\_stdf\_to\_dataframe(stdf\_file\_path):

"""

Parses an STDF file and extracts relevant records into a pandas DataFrame.

Focuses on PTR records for test results.

You might need to adjust this based on what records you need.

"""

data = []

try:

with open(stdf\_file\_path, 'rb') as f:

for record in V4.records\_from\_file(f):

if isinstance(record, V4.PTR):

# Extract key information from PTR record

data.append({

'FILE\_PATH': stdf\_file\_path, # Keep track of the source file

'REC\_TYP': record.REC\_TYP,

'REC\_SUB': record.REC\_SUB,

'HEAD\_NUM': record.HEAD\_NUM,

'SITE\_NUM': record.SITE\_NUM,

'TEST\_NUM': record.TEST\_NUM,

'TEST\_NAM': record.TEST\_NAM if hasattr(record, 'TEST\_NAM') else None,

'HARD\_BIN': record.HARD\_BIN if hasattr(record, 'HARD\_BIN') else None,

'SOFT\_BIN': record.SOFT\_BIN if hasattr(record, 'SOFT\_BIN') else None,

'RESULT': record.RESULT, # The test result (e.g., value, pass/fail code)

'TEST\_FLG': record.TEST\_FLG, # Test flags (e.g., pass/fail status)

'RTN\_ICNT': record.RTN\_ICNT, # Retest count - very important for your case

'X\_COORD': record.X\_COORD if hasattr(record, 'X\_COORD') else None,

'Y\_COORD': record.Y\_COORD if hasattr(record, 'Y\_COORD') else None,

# Add other relevant fields from PTR or other records like PRR if needed

# For PRR (Part Results Record), you might also want to track PART\_ID

# To get PART\_ID, you'd need to link PRR records to PTRs.

# This example simplifies by only getting PTRs.

})

# You might need to process PRR records to get PART\_ID

# For simplicity, we'll assume HEAD\_NUM and SITE\_NUM are enough for part identification for this example.

# If PART\_ID is crucial, you'll need to store PRR data and merge it based on sequence.

# Or, if your PTRs contain unique part IDs, use that.

except Exception as e:

print(f"Error processing {stdf\_file\_path}: {e}")

return pd.DataFrame(data)

def merge\_stdf\_files\_final\_results(stdf\_files\_paths, output\_csv\_path="merged\_final\_results.csv"):

"""

Merges multiple STDF files, keeping only the final test result for each

part/test combination.

Args:

stdf\_files\_paths (list): A list of paths to the STDF files, ordered

from oldest to newest (first is largest, last is smallest).

output\_csv\_path (str): The path to save the merged results as a CSV.

"""

all\_dataframes = []

print("Parsing STDF files...")

for file\_path in stdf\_files\_paths:

print(f" - Parsing {file\_path}")

df = parse\_stdf\_to\_dataframe(file\_path)

if not df.empty:

df['SOURCE\_FILE\_INDEX'] = stdf\_files\_paths.index(file\_path) # Helps in ordering later

all\_dataframes.append(df)

if not all\_dataframes:

print("No data parsed from any STDF files.")

return

# Concatenate all dataframes

merged\_df = pd.concat(all\_dataframes, ignore\_index=True)

print(f"Total records after initial merge: {len(merged\_df)}")

# Sort to ensure later files' results come after earlier ones for the same part/test.

# The order of sorting keys is crucial for correct deduplication.

# We sort by:

# 1. Part identifier (HEAD\_NUM, SITE\_NUM - if PART\_ID is available, use it!)

# 2. Test identifier (TEST\_NUM, TEST\_NAM)

# 3. Source file index (to prioritize later files in case of exact duplicates in earlier files)

# 4. Retest count (RTN\_ICNT) - higher retest count indicates a later run for the same part/test within a file or across files if the part was re-tested.

merged\_df.sort\_values(

by=['HEAD\_NUM', 'SITE\_NUM', 'TEST\_NUM', 'TEST\_NAM', 'SOURCE\_FILE\_INDEX', 'RTN\_ICNT'],

inplace=True,

ascending=[True, True, True, True, True, True] # Ascending for all to ensure later tests are at the bottom

)

# Drop duplicates, keeping the last occurrence.

# 'HEAD\_NUM', 'SITE\_NUM', 'TEST\_NUM', 'TEST\_NAM' define a unique test for a unique part.

# Because of the sorting, `keep='last'` will select the entry from the latest file (highest SOURCE\_FILE\_INDEX)

# and the highest RTN\_ICNT for that part/test combination.

final\_results\_df = merged\_df.drop\_duplicates(

subset=['HEAD\_NUM', 'SITE\_NUM', 'TEST\_NUM', 'TEST\_NAM'],

keep='last'

)

print(f"Total records after deduplication (final results): {len(final\_results\_df)}")

# Clean up and save

final\_results\_df = final\_results\_df.drop(columns=['FILE\_PATH', 'SOURCE\_FILE\_INDEX'])

final\_results\_df.to\_csv(output\_csv\_path, index=False)

print(f"Merged final results saved to {output\_csv\_path}")

# --- Example Usage ---

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

# Create dummy STDF files for demonstration

# In a real scenario, you would replace these with your actual file paths.

# File 1 (largest): Initial run, some failures

# File 2 (smaller): Retest of some failures from File 1

# File 3 (smallest): Retest of some remaining failures from File 2

# Dummy data for demonstration

# For simplicity, we'll manually create some STDF records.

# In a real scenario, these would be generated by your tester.

# Helper to write dummy STDF files

def write\_dummy\_stdf(file\_path, records):

with open(file\_path, 'wb') as f:

for rec in records:

f.write(rec.to\_bytes())

# Dummy records for File 1

# Part 1, Site 0: Test 1 (Pass), Test 2 (Fail)

# Part 2, Site 1: Test 1 (Pass), Test 2 (Pass)

dummy\_records\_1 = [

V4.FAR(CPU\_TYPE=2, STDF\_VER=4),

V4.MIR(LOT\_ID='LOT\_ABC', NODE\_NAM='Tester1', TSTR\_TYP='ATE', TEST\_TIM=1678886400),

V4.SDR(HEAD\_NUM=1, SITE\_GRP=0, SITE\_NUM=1), # Dummy site information

V4.PRR(HEAD\_NUM=1, SITE\_NUM=0, PART\_FLG=0x00, NUM\_TEST=2, HARD\_BIN=1, SOFT\_BIN=1), # Part 1 (Site 0)

V4.PTR(HEAD\_NUM=1, SITE\_NUM=0, TEST\_NUM=1, TEST\_FLG=0x00, RESULT=10.5, RTN\_ICNT=0), # Part 1 Test 1 (Pass)

V4.PTR(HEAD\_NUM=1, SITE\_NUM=0, TEST\_NUM=2, TEST\_FLG=0x80, RESULT=1.2, RTN\_ICNT=0), # Part 1 Test 2 (Fail)

V4.PRR(HEAD\_NUM=1, SITE\_NUM=1, PART\_FLG=0x00, NUM\_TEST=2, HARD\_BIN=1, SOFT\_BIN=1), # Part 2 (Site 1)

V4.PTR(HEAD\_NUM=1, SITE\_NUM=1, TEST\_NUM=1, TEST\_FLG=0x00, RESULT=20.1, RTN\_ICNT=0), # Part 2 Test 1 (Pass)

V4.PTR(HEAD\_NUM=1, SITE\_NUM=1, TEST\_NUM=2, TEST\_FLG=0x00, RESULT=5.0, RTN\_ICNT=0), # Part 2 Test 2 (Pass)

V4.MRR(FINL\_PMOD=0)

]

stdf\_file\_1 = "lot\_test\_1.stdf"

write\_dummy\_stdf(stdf\_file\_1, dummy\_records\_1)

# Dummy records for File 2 (retest of Part 1, Test 2)

# Part 1, Site 0: Test 2 (Pass) - this is the retest, so RTN\_ICNT should be > 0 or it's a new run

dummy\_records\_2 = [

V4.FAR(CPU\_TYPE=2, STDF\_VER=4),

V4.MIR(LOT\_ID='LOT\_ABC', NODE\_NAM='Tester2', TSTR\_TYP='ATE', TEST\_TIM=1678886500),

V4.SDR(HEAD\_NUM=1, SITE\_GRP=0, SITE\_NUM=1),

V4.PRR(HEAD\_NUM=1, SITE\_NUM=0, PART\_FLG=0x00, NUM\_TEST=1, HARD\_BIN=1, SOFT\_BIN=1), # Part 1 (Site 0)

V4.PTR(HEAD\_NUM=1, SITE\_NUM=0, TEST\_NUM=2, TEST\_FLG=0x00, RESULT=1.5, RTN\_ICNT=1), # Part 1 Test 2 (Retest, now Pass)

V4.MRR(FINL\_PMOD=0)

]

stdf\_file\_2 = "lot\_test\_2.stdf"

write\_dummy\_stdf(stdf\_file\_2, dummy\_records\_2)

# Dummy records for File 3 (a different part retest)

# Part 3, Site 0: Test 1 (Pass)

dummy\_records\_3 = [

V4.FAR(CPU\_TYPE=2, STDF\_VER=4),

V4.MIR(LOT\_ID='LOT\_ABC', NODE\_NAM='Tester3', TSTR\_TYP='ATE', TEST\_TIM=1678886600),

V4.SDR(HEAD\_NUM=1, SITE\_GRP=0, SITE\_NUM=1),

V4.PRR(HEAD\_NUM=1, SITE\_NUM=2, PART\_FLG=0x00, NUM\_TEST=1, HARD\_BIN=1, SOFT\_BIN=1), # Part 3 (Site 2)

V4.PTR(HEAD\_NUM=1, SITE\_NUM=2, TEST\_NUM=1, TEST\_FLG=0x00, RESULT=7.8, RTN\_ICNT=0), # Part 3 Test 1 (Pass)

V4.MRR(FINL\_PMOD=0)

]

stdf\_file\_3 = "lot\_test\_3.stdf"

write\_dummy\_stdf(stdf\_file\_3, dummy\_records\_3)

# List of STDF files in the order they were generated (or re-tested)

# This order is crucial for correctly identifying "final" results.

stdf\_files = [stdf\_file\_1, stdf\_file\_2, stdf\_file\_3]

output\_csv = "merged\_final\_stdf\_results.csv"

merge\_stdf\_files\_final\_results(stdf\_files, output\_csv)

print("\nContent of the merged CSV file:")

print(pd.read\_csv(output\_csv))

# Clean up dummy files

os.remove(stdf\_file\_1)

os.remove(stdf\_file\_2)

os.remove(stdf\_file\_3)

Explanation:

1. parse\_stdf\_to\_dataframe(stdf\_file\_path) function:
   * This function takes an STDF file path as input.
   * It uses V4.records\_from\_file() from Semi\_ATE.STDF to iterate through the records in the STDF file.
   * It specifically targets V4.PTR (Parametric Test Record) records, as these contain the actual test results. You might also want to extract data from V4.PRR (Part Results Record) if PART\_ID is defined there and you need it for unique part identification.
   * For each PTR record, it extracts key information like HEAD\_NUM (test head number), SITE\_NUM (test site number), TEST\_NUM (test number), TEST\_NAM (test name), RESULT, TEST\_FLG (pass/fail status), and crucially, RTN\_ICNT (retest count).
   * RTN\_ICNT is important as it indicates if a test for a particular part has been re-run. A higher RTN\_ICNT generally implies a later retest.
   * It adds a FILE\_PATH column to track the source file.
   * All extracted data is accumulated into a list of dictionaries, which is then converted into a pandas DataFrame.
2. merge\_stdf\_files\_final\_results(stdf\_files\_paths, output\_csv\_path) function:
   * Takes a list of STDF file paths and an output CSV path.
   * Parsing: It iterates through each stdf\_file\_path, calls parse\_stdf\_to\_dataframe to get a DataFrame for each file, and adds a SOURCE\_FILE\_INDEX column. This index is critical for maintaining the processing order of your files (since later files represent more recent tests).
   * Concatenation: All individual DataFrames are concatenated into a single large DataFrame (merged\_df).
   * Sorting: This is the most crucial step for deduplication. The merged\_df is sorted by:
     + HEAD\_NUM, SITE\_NUM: To group all tests for a specific physical part. If your STDF files contain PART\_ID in PRR records and you can link it to PTR records, using PART\_ID would be even more robust.
     + TEST\_NUM, TEST\_NAM: To identify a specific test within a part's test sequence.
     + SOURCE\_FILE\_INDEX: This ensures that results from later STDF files (which contain re-tests) appear *after* results for the same part/test from earlier files.
     + RTN\_ICNT: If a part/test was re-tested *within* the same STDF file (less common for your scenario, but good practice), a higher RTN\_ICNT would indicate the later retest.
   * Deduplication: merged\_df.drop\_duplicates(subset=[...], keep='last') is used.
     + subset: Defines the columns that uniquely identify a "test for a part" (e.g., HEAD\_NUM, SITE\_NUM, TEST\_NUM, TEST\_NAM).
     + keep='last': Because of the careful sorting, keep='last' will retain the entry that was processed *last* for each unique part-test combination. This means it will keep the result from the latest STDF file and the highest RTN\_ICNT if there were multiple retests for that specific test within a single file or across files.
   * Saving: The final\_results\_df is then saved to a CSV file.

Important Considerations and Potential Enhancements:

* Part Identification: The current example uses HEAD\_NUM and SITE\_NUM to identify a part. In a real-world scenario, you will very likely need to parse PRR (Part Results Record) records to get the unique PART\_ID and potentially WAFER\_ID if tests are conducted at the wafer level. Then, you'd need to associate the PTR records with their corresponding PRR (usually by HEAD\_NUM and SITE\_NUM and order of appearance in the file) to get the complete part identification.
* Test Identification: The combination of TEST\_NUM and TEST\_NAM is generally good for identifying a unique test.
* Retest Logic (RTN\_ICNT): The RTN\_ICNT field in PTR records is crucial. If a part failed and was re-tested, RTN\_ICNT will increment. Sorting by this helps ensure the latest result is kept even if multiple retests appear in the same file.
* Other Record Types: Depending on your analysis needs, you might want to extract information from other STDF record types (e.g., PIR for Part Information, TSR for Test Summary, HBR for Hardware Bin, SBR for Software Bin). The Semi-ATE library provides access to all these.
* Error Handling: The parse\_stdf\_to\_dataframe includes basic error handling, but for production code, you might want more robust logging and error management.
* Memory Usage: For very large STDF files or many files, loading all data into a single pandas DataFrame might consume significant memory. If this becomes an issue, consider:
  + Processing in chunks: Process records in smaller batches if the Semi-ATE library supports it for PTR records.
  + Database: Load data into a temporary database (like SQLite) and perform the merge/deduplication with SQL queries, which are optimized for large datasets.
  + Parquet/HDF5: Save intermediate DataFrames to Parquet or HDF5 format, which are more efficient for columnar data storage and retrieval, and then load/merge.
* Custom Retest Definitions: Your specific retest methodology might have nuances. For example, if a part is sent to a *different* tester for retest, or if the "final" result is determined by something other than the latest timestamp or retest count, you'll need to adjust the sorting and deduplication logic accordingly.

This code provides a solid foundation for merging your STDF files and extracting the final test results. Remember to adapt the part and test identification logic to precisely match how these are recorded in your specific STDF files.