CMPT 473 Assignment 1

Natalie Woods

lib/kernel/streamutils/deletion.cpp

Arshdeep Mann

lib/kernel/scan/line_span_generator.cpp

Timothy Chua

lib/kernel/streamutils/sorting.cpp

Brian Del Carpio

lib/kernel/streamutils/pdep_kernel.cpp

Proposal

Group 67 has conducted Assignment 1 using the open-source Parabix-Devel project, with the intention of further utilizing and analyzing the repository for the duration of the course. Following direct recommendations from Professor Cameron, our group opted to use this project as it was outfitted with a large and robust test suite with a substantial quantity of contributions. The nature of Parabix as a framework for text processing and rapid expression searching made it an ideal and accessible choice for students. As such, the group proposes for the continued use of Parabix in the foreseeable future.

Evaluations and assignment objectives are met using CSIL Linux resources. The project itself utilizes C++, making a minimal amount of reading and writing in C++ a necessity for mutation testing. Automation has been handled with shell scripting; Python also being used at student discretion. The project's clean build time on a CSIL connection ran upwards of 20 minutes, while incremental build time would fall under 1 minute. Execution for the full test suite would regularly exceed 500 seconds per mutant. Assuming similar objectives are to be met in subsequent assignments, it is proposed these tools and conditions continue to be used by our group for additional coursework.

streamutils/deletion.cpp:

Test Suite Application and Results

Mutation Operators Used:

Loop start offset changes
- for (i = 0; ...) -> for (i = 1; ...) (M04, M08, M14, M25, M50)

Loop step mutations
- i++ -> i-- (M06, M13)

Shift direction change
- 1 << i -> 1 >> i (M07)

Multiply/divide swaps
- bitBlockWidth / fieldWidth → bitBlockWidth * fieldWidth (M29,

Add/Subtract swaps
- numFields - 1 -> numFields + 1 (M16)

Comparison swaps
- >= → > (M64)
- == ↔ != (M18, M45, M73, M79)
- <= → < (M81)

Logic connective swaps
- && → || (M34, M38, M57)

AND ↔ OR swaps
- CreateAnd → CreateOr (M58, M60, M71, M90, M92, M100)

Wrong variable used as an index
- mask[i] → mask[j] (M12)

Results:

Many mutants were killed quickly by the following tests:
- test_emptyprogram
- greptest
- test_bytefilterandspread

Status	Count	Percentage
Killed	76	75.2%
Survived	23	22.8%
Broken	2	2.0%
TOTAL	**101**	**100%**

Mutant results per routine:

Routine	Total	Killed	Survived	Effectiveness
GenerateMultiBlockLogic	37	27	10	73%
ByteFilterByMaskKernel	27	24	3	88.9%
SwizzledDeleteBy PEXTkernel	17	13	4	76.5%
SwizzledBitstreamv CompressByCount	12	10	2	83.3%
Others (6 routines)	23	16	7	69.6%

TOTAL Including: 2 Broken + 1 Equivalent	**116**	**90**	**29**	**79.6%**

Analysis Of Surviving Mutants

1. **Mutant M001 -** Not equivalent: Changes output
   - (shift = 1; ..) -> for (shift = 2; ..)

     Could add a test that has a deletion mask pattern that needs the very first shift step(shift = 1) to correctly move bits.

2. **Mutant M003 -** Not equivalent: Changes output
   - (... lookright *= 2) -> for (... lookright += 2)

     Could add a test that uses a deletion mask with multiple spaced deletions. Then compare output against a reference implementation.

3. **Mutant M005** - Not equivalent: Loop never runs
   - for (unsigned i = 0; i < mv.size(); …) -> for (unsigned i = 0; i > mv.size(); …)

     Could create a test where the input has a known pattern and could verify the output matches.

4. **Mutant M007** – Not equivalent: Changes output
   - unsigned shift - 1 << i; -> unsigned shift - 1 >> i;

     Could add a test that creates a deletion mask that forces data to move across several positions. Then check the output after each iteration.

5. **Mutant M045** – Equivalent
   - if (getStride() != b.getBitBlockWidth() -> if (getStride() != b.getBitBlockWidth()

The body in this if conditon is: numOfBlocks = b.CreateShl(numOfStrides, b.getSize(floor_log2(getStride()/b.getBitBlockWidth())));

In the original code if the statement is false the body isnt executed. In the mutated code, if the condition is true, the body does execute which is a shift. But getStride()/b.getBitBlockWidth() = 1 and floor_log2(1) = 0

So nothing would happen as it would result in shifting left by 0

Methods and Observations

I produced the mutants manually but for next time I would use an existing mutation generation framework that already knows C++ syntax and wouldn't generate nonsense mutants. To evaluate the mutants, I wrote a script that would automatically apply each mutation, rebuild the project, and execute the full test suite against each mutant. One of the main observations from this experiment was the test effectiveness varied between routines. This also depended on the mutants I chose as well. For example, routines that used bit masking had several surviving mutants. It's also important to note that surviving does not always indicate poor testing. There could be several equivalent mutants that don’t affect the output and functionally identical to the original code. I also had a couple mutants that were classified as broken due to compilation failures so those were subtracted from the effectiveness score as they were never evaluated.

scan/line_span_generator.cpp

Test Suite Application and Results

The test_scan test suite was applied to all generated mutants. In total, 93 mutants were tested, of which 67 were killed, 23 survived, and 3 resulted in timeouts. No compile errors were observed. Effectiveness for generateProcessingLogic was 38/38 (100%), while generateMultiBlockLogic achieved 29/(29+23), or 55.8%. Overall mutation score was 67/93 (72.0%).

Routine	Total	Killed	Survived	Timeout	Effectiveness
generateMultiBlockLogic	55	29	23	3	52.7%
generateProcessingLogic	38	38	0	0	100%
TOTAL	**93**	**67**	**23**	**3**	**72.0%**

Analysis Of Surviving Mutants

1. **Mutant M023**
   Occurs in generateMultiBlockLogic at line 71 and replaces an equality comparison with an unsigned less-or-equal comparison when checking whether the number of available line numbers is zero. Because the value is unsigned, the condition <= 0 is only true when the value is exactly zero, making the mutant semantically equivalent to the original. This mutant is therefore equivalent and should not be killed.

2. **Mutant M038**
   Located at line 79 in generateMultiBlockLogic, replaces a < comparison with a >= comparison when checking whether additional spans remain. The original condition ensures that spans are still available, while the mutant represents the opposite logic. This value is used in the loop continuation condition, so the mutation can allow the loop to

continue after all spans are exhausted, potentially causing incorrect behavior or out-of-bounds access. This mutant is non-equivalent. A test that exhausts the span stream before the line-number stream (or vice versa) and verifies correct termination would kill this mutant.

3. **Mutant M048**
   at line 92 in generateMultiBlockLogic, changes a branch condition from equality to unsigned less-or-equal when deciding whether to use or skip a span. An earlier assertion guarantees that lineNumVal is always greater than or equal to spanIndex, meaning the condition can only be true when the two values are equal. As a result, the mutated condition produces identical behavior to the original. This mutant is equivalent and does not require additional testing.

4. **Mutant M075**
   found at line 73 in generateMultiBlockLogic, replaces a logical OR with a logical AND when computing whether no input streams are available. The original logic exits when either the line-number stream or the span stream is empty, while the mutant exits only when both are empty. This allows execution to continue when exactly one stream has been exhausted, which can lead to invalid reads or incorrect output. This mutant is non-equivalent. A test where one stream is empty and the other is non-empty would expose the fault and kill the mutant.

5. **Mutant M077**
   at line 80 in generateMultiBlockLogic, changes a logical AND to a logical OR in the continuation check returned by GenerateContinueCheck. The original requires that both streams have remaining elements in order to continue, while the mutant allows continuation when either stream still has data. This can result in the loop continuing after one stream is exhausted, again risking out-of-bounds access or incorrect results. This mutant is non-equivalent, and a test that exhausts one stream before the other and checks for correct termination would kill it.

Methods and Observations

Mutation testing shows full coverage of generateProcessingLogic but significantly weaker coverage of generateMultiBlockLogic, where most surviving mutants are concentrated. These survivors primarily arise from relational and logical operator mutations in boundary and loop-continuation conditions, indicating insufficient testing of edge cases.

Mutants were generated using a Python script that applied standard mutation operators (AOR, ROR, CR, LOR), with one mutation per location. A shell script executed the test suite against each mutant and recorded outcomes in mutation_results.txt. Each mutant and its associated diff were logged in mutations_log.txt. Both generation and execution were fully automated.

Equivalent mutants typically resulted from mutations that preserved semantics due to unsigned comparisons or strong invariants enforced earlier in the code. Many non-equivalent surviving mutants highlight gaps in test coverage, particularly for empty-stream and asymmetrically sized input scenarios. Mutations that caused infinite loops, such as loop-increment changes, were skipped or resulted in timeouts. Overall, this study demonstrates that targeted edge-case tests are essential for improving mutation coverage in stream-processing and loop-control logic, and that automation is crucial for managing a large mutant set efficiently.

streamutils/sorting.cpp

Test Suite Application and Results

The full test suite was applied to 100 prepared mutants. Code reachability was a primary concern, prompting the usage of all 22 tests regardless of obvious or immediate interaction with the target file. Initially, attempts were made to test the mutations using meta_tests, test_bit_movement, index_test, and test_bytefilterandspread exclusively. The decision was made to proceed with a full "make check" following exceedingly low effectiveness numbers, and lack of confidence in both provided and created sanity checks.

Routine	**Total**	**Killed**	**Survived**	**Broken**	**Effectiveness**
AdjustsRunsAndIndexes:: generatePabloMethod	31	12	2	17	87.5%
BitonicSort:: generatePabloMethod	20	20	0	0	100%
BitonicCompareStep:: generatePabloMethod	15	12	1	2	92.3%
AppendStreamSets:: generatePabloMethod	15	14	0	1	100%
Others (3 routines)	17	14	0	3	100%
TOTAL Including: 2 equivalents	**98**	**72**	**3**	**23**	**67.7%**

Originally there was interest in generating the mutations using Mull, of which has been acknowledged in class. However, as I conducted this assignment using CSIL, I had resorted to manual creation of “diff” files. Efforts were alleviated by focusing on producing “diff” files for operators with multiple swapping options.

Arithmetic operators present in the original code were swapped to produce mutations.

E.g.) + → -, *, /, %

All logic operators present in the original code were swapped to produce mutations.

E.g.) > → >=. <, <=, !=, ==

Methods from Pablo AST were also considered as targets for mutation, with there being switches between createAnd and createOr. Full documentation of operators can be found in the corresponding “unified diff” file.

A shell script was written to apply the mutations against the test suite, with mutations running on an average of 525 seconds. Results were parsed from a text file and manually compiled for the purpose of this report.

Analysis Of Surviving Mutants

1. **Mutants M003, M005**
   Change the main loop of the AdjustRunsAndIndexes void function. The main loop is responsible for iterating over the length of an expression of a sequence index for sorting between ordered and unordered runs. Mutation 3 provides the opportunity for the main loop to go out of bounds. Mutation 5 does not have this problem, as there are no statements within the loop that can cause the variable lgth exceed an increment of 1 to skip past maxLgth. While mutation 5 is arguably an equivalent mutation, mutation 3 should assert that lgth does not equal maxLgth.

2. **Mutants M008, M010**
   Edit the nested loop of the AdjustRunsAndIndexes void function. This loop specifically sets the i’th value of SeqIndexAhead, used for determining the start of a run. Maintaining iterations over SeqIndexAhead in increments of 1 within SeqIndex.size() is required for this function. Mutation 10 works as an equivalent mutation in that there are no statements causing the loop to increment beyond SeqIndex.size(). However, an assertion is necessary for mutation 8 given its inclusive nature, with a check for i != SeqIndex.size().

3. **Mutant M016**
   This statement is used for determining element assignment to split bit subsequences. The mutation makes the condition inclusive, creating a case for one element to join the incorrect region, possibly going out of bounds. Additional testing should be introduced, asserting against current region capacity.

Methods and Observations

Mutations applied to the target file produced 5 survivors and 72 kills. Earlier results suggested there were 8 survivors, but following adjustments to extend the threshold of timeouts on the test scripts there were three additional kills secured by both nfd_test and nfc_test. These two tests are also responsible for the majority of the killed mutants. Two equivalent mutations were identified, reducing the survivor count to 3. The remaining 23 mutants were deemed broken, including timeouts and build failures. Salvaging broken mutations was determined to be outside the scope of this assignment, with the occurrence considered as an expected outcome and part of the quality assurance experience. Test suite effectiveness yielded an average of 89.9%.

streamutils/pdep_kernel.cpp

Test Suite Application and Results

The full test suite was applied for each mutant created, with a focus on mathematical and logical operators on various routines in the file. Overall, the test suite killed 65 out of the 96 mutants tested meaning a 67.7% effectiveness rate. However, with 23 of the mutants being equivalents, primarily from assertions or safe-off-by-one errors, the test suite reached and impressive 91.5%. The tests effectiveness is broken down further in the figure below.

Routine	**Total**	**Killed**	**Survived**	**Timeout**	**Effectiveness**
ByteSpreadByMaskKernel	19	13	5	1	68.4%
StreamExpandKernel	13	9	4	0	69.2%
MergeByMask	10	7	2	1	70.0%
StreamMergeKernel	9	9	0	0	100%
Others (15 routines)	45	27	18	0	60.0%
TOTAL	**96**	**65**	**29**	**2**	**67.7%**

Analysis Of Surviving Mutants

1. **Mutant M002** – line 109
   This mutant replaced a logical AND with an OR operation in line 109 on the routine StreamMergeKernel. This mutant still produces a functionally correct result because this

if statement is used to improve performance between two methods that perfom the same merge action.

2. **Mutant M058** – line 106
   This mutant survives because the change ensures that the error is only reported when both conditions are wrong. This results in a weaker check than the original. Although this mutation could get caught by adding assertions, it is not needed since the original tests have stricter rules.

3. **Mutant M040** – line 63
   This mutant inverts a != to a ==. The original code calculates an offset when the input and output sizes do not match. The mutant flips this to calculate the offset only when sizes match, making the offset equal to zero. This mutant is not getting caught probably because the tests only run cases where the sizes are equal, hence to catch it tests with different sizes should be added.

4. **Mutant M062** – line 55
   The code mutated decides between two paths for the same work, since the test suite does not check for which path is taken, only that the answer is right, the mutant survives. To detect and kill this mutant, a mock or stub should be used to detect which path is taken based on the conditions in the test

5. **Mutant M018** – line 1033
   This mutant affects how a string is built and flips the decision between appending "Before" or "After" to it. The mutant survives, likely because the tests are not focused on building the right string. However a silent bug like this could have large implications when a human tries to read output. Tests should be added to assert whether the string does agree with the expected output in each situation.

Methods and Observations

The mutants were produced with a python script to detect mathematical and logical symbols. The script mutated a single symbol, built, and ran the test suite, logged the output and the difference, for inspection, and finally reverted the changes to keep a consistent original copy of the pdep_kernel.cpp file. The biggest observations was how mutant testing is able to find subtle bugs that a developer could overlook with out batting an eye. For example the surviving test on the string operation could completely invalidate data a human would be looking into. However, it is also important to keep in mind that not all surviving mutants mean there is bad tests, out of the 31 surviving mutants only 10 mutants were not equivalent.