KLEE is a tool designed to increase the code coverage and find more bugs as compared to manual testing of the code. The results seem to be pretty good in both the aspects, when it was tested on COREUTILS and other UNIX libraries. The most important part according to me is about eliminating false positives (although in practice, with non-determinism in code, it has result into false positives). Generating test cases quickly and automatically is a much better approach than manually testing the code and also, doing symbolic execution helps examining the code in depth and also makes it less dependent on the environment it is tested on (instead of with fixed values, where the code parsing can end abruptly).

First compile the code to bytecode using LLVM and then run KLEE on this generated bytecode. In the HW practice problems, we saw what commands to use and how fast KLEE was. On page 3, they explain how KLEE works using the MINIX's tr code and how KLEE finds a buffer flow error and generates appropriate test case for it. Another nice feature I find in this tool is we can control how many fails we want before termination and the type of arguments we want to test it on.

In case of KLEE, storage locations refer to expressions instead of raw data (because it is symbolic execution) and leaves of the expression are symbolic variables or constants. Conditional branches take a Boolean expression. After executing each statement, it updates the Path condition and program counter until it hits a bug or an exit command. When KLEE finds that both paths can be explored, it clones the state to explore both paths and updates the instruction pointer and path condition respectively. Dangerous operations like checking the divisor to be 0 is handled like normal branches (it keeps following the false path after negating the condition until it hits an error or exit condition confirming that it can never happen). KLEE checks for address is in bounds for load and store operations. In case when a dereference pointer can refer to N objects, KLEE clones the state N times and appropriately applies the constraints.

The several query optimizations done to reduce the cost of constraint solving are: 1. Expression rewriting, 2. Constraint set simplification 3. Implied value concretization 4. Constraint independence 5. Counter-example cache.

The number of states in KLEE grow really fast and it becomes crucial to select which state to execute next. KLEE does that using 2 search heuristics. 1. Random path selection 2. Coverage optimized search tree.

Conditions that KLEE can handle:

1. Division (including division by zero)
2. Major integer arithmetic (except for large unroll for multiplications)
3. Generating test cases and high coverage even in case of path explosion
4. Less chances for false positives in a static environment
5. Able to reach deeper in code because of optimizations performed on path conditions and optimizations on the constraints.

Conditions that KLEE cannot handle:

1. Large loop unrolls/ high recursive calls.
2. If the disk quota exceeds or the code depends a lot on the environment (like inputs and setting of parameters is highly dependent)
3. No support for floating-point so integer division operations can sometimes be not tracked.
4. KLEE does explore the path which can never be covered in the code but doesn't generate tests for such paths which in a dynamic setting can lead to false positives.

Along with this document, I submitted 3 following:

The examples for where KLEE was successful:

1. Absolute value function.
2. Some linear interpolation of 2 integers.
3. Finding a maximum of 2 numbers.

The examples where KLEE failed:

1. Infinite loop (KLEE doesn't really terminate well and spends a lot of trying to solve this).
2. Integer division resulting in a float value
3. A long recursion or loop unroll with multiplication operation. (needs to be highly constraint to terminate i.e. needs a strict upper bound and lower bound). When I experimented with upper bound missing or a lower bound missing, it didn't terminate for a long time.