4.SYMBOLIC EXECUTION

Symbolic execution is a software testing technique that is useful to aid

the generation of test data and in proving the program quality.

Steps to use Symbolic Execution:

- The execution requires a selection of paths that are exercised bya set of data values. A program, which is executed using actual data, results in the output of a series of values.
- In symbolic execution, the data is replaced by symbolic values with set of expressions, one expression per output variable.
- The common approach for symbolic execution is to perform an analysis of the program, resulting in the creation of a flow graph.
- The flowgraph identifies the decision points and the assignments associated with each flow. By traversing the flow graph from an entry point, a list of assignment statements and branch predicates is produced.

Issues with Symbolic Execution:

- Symbolic execution cannot proceed if the number of iterations in the loop is known.
- The second issue is the invocation of any out-of-line code or module calls.
- Symbolic execution cannot be used with arrays.
- The symbolic execution cannot identify of infeasible paths.

Symbolic Execution Application:

- Path domain checking

- Test Data generation
- Partition analysis
- Symbolic debugging

MODEL CHECKING

Model checking is the most successful approach that's emerged for verifying requirements.

The essential idea behind model checking is A model-checking tool accepts system requirements or design (called models) and a property(called specification) that the final system is expected to satisfy.

The tool then outputs yes if the given model satisfies given specifications and generates a counterexample otherwise. The counterexample details why the model doesn't satisfy the specification. By studying the counterexample, you can pinpoint the source of the error in the model, correct the model, and try again. The idea is that by ensuring that the model satisfies enough system properties, we increase our confidence in the correctness of the model. The system requirements are called models because they represent requirements or design.

So what formal language works for defining models? There's no single answer, since requirements (or design) for systems in different application domains vary greatly.

For instance, requirements of a banking system and an aerospace system differ in size, structure, complexity, nature of system data, and operations performed.

In contrast, most real-time embedded or safety-critical systems are control-oriented rather than data-oriented—meaning that dynamic behavior is much more important than business logic (the structure of and operations on the internal data maintained by the system).

Such control-oriented systems occur in a wide variety of domains: aerospace, avionics, automotive, biomedical instrumentation, industrial automation and process control, railways, nuclear power plants, and so forth. Even communication and security protocols in digital hardware systems can be thought of as control oriented.

For control-oriented systems, finite state machines (FSM) are widelyaccepted as a good, clean, and abstract notation for defining requirements and design. But of course, a “pure” FSM is not adequate for modeling complex real-life industrial systems. We also need to:

- be able to modularize the requirements to view them at different levels of detail
- have a way to combine requirements (or design) of components
- be able to state variables and facilities to update them in order to use them in guards on transitions.

In short, we need extended finite state machines (EFSM) . Most model checking tools have their own rigorous formal language for defining models, but most of them are some variant of the EFSM.