

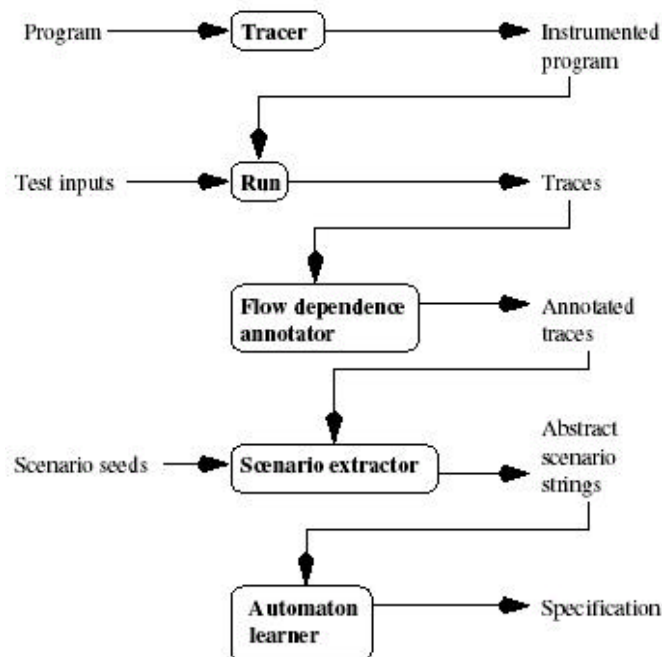
Mining Specifications

Authors: Glenn Ammons, Rastislav Rodík, James R. Larus
Summarized by: Feng Peng

Basic Idea:

An executing program is usually “almost” right, i.e., if we execute the program repeatedly, most of the results should be correct. If the program is written according to an API or ADT, which are defined using some specification, its running behavior can be used as a very close approximation of the specification. Therefore, a mining program can identify the common behavior such as temporal and data dependence of the program by examining the traces of multiple executions. The result is an FSM (finite state machine) that depicts the behaviors of the program. Programmer, or automatic verification tools, can examine the FSM to find bugs or deficits of the specification.

System Architecture:



The mining program works as follows.

1. A **tracer** program instruments the program so that executing the program produces a trace of its interactions with an API or ADT. Basically the user has to modify the program so that each function call to the API/ADT is replaced by an instrumented version that will record the value of the parameters when the function is called and returned.
2. Execute the programs to get traces of the running programs. The trace is a sequence of **interactions**. Each interaction is a named function call to the API/ADT together with the value of the arguments.
3. A **flow dependence annotator** first applies a STM (state transition model, specified by human experts) to the traces to get the annotated traces in the form of **PDG** (program dependence graph). It then applies type inference to the arguments of the interactions in the PDG. An argument is typed if and only if it is involved in the flow dependence.

4. A **scenario extractor** uses a user-specified interaction seed to find a scenario that contains the seed together with some of its ancestors and descendants. The number of ancestors and descendants is restricted by a parameter to make the learning process later tractable. A **scenario** is a small set of independent interactions that for any two of them, the interactions between them (in the dependence graph) are also in the scenario.
5. The extracted scenarios are then **simplified** (remove useless, i.e., untyped attributes in the interactions) and **standardized** (assign each a interaction in the scenario a letter and convert the scenario into a string that can be used in **PFSA learner**). Note that reordering (based on the fact that each scenario is a DAG so that we can put different scenarios of the same DAG together) and typing (remove duplicated scenarios that are only different in the name of variables) are used in the letter assignment so that similar scenarios get similar scenario strings. For example, a naïve algorithm can convert the string using all dependence-preserving permutations and use the first one of them in lexicographic order.
6. The standardized scenario strings are then fed into a **PFSA (probabilistic finite state automaton)** learner to get the automaton for the specification. In the first step, the PFSA learner builds a **weighted retrieval tree**. The nodes in the tree are labeled with the interaction letter. The edges in the tree are labeled with the frequency of the (parent, child) substring appears in the training set. A path in the tree is corresponding to a string in the training set. In the second step, similar nodes (we can also call them states if we treat the tree as a PFSA) are merged together to reduce the number of states in the automaton. A **corer** trims the result PFSA to remove the infrequent (low heat) edges in it. The **heat** of an edge is defined as its likelihood of being traversed while generating a string from the PFSA, which can be computed using Markov-chain.

Verification Tool:

Though the authors do not implement automated verification tool based on the idea, they proposed two methods to construct verification tool. The first method is to iterate all the seeds, for each seed, try to find a scenario to satisfy the specification while running the program. If the specification is correct, we can at least find one scenario that satisfies it. If we have reached the maximum size of the scenario and we cannot find one scenario that satisfies, the program reports error. The other method is going from the other direction. It generates a trace automaton from the specification. The automaton generates all possible traces (and nothing else) that satisfy the specification. The verification tool then searches all the traces to see if it can find one that cannot generated by the automaton and reports an error if one is found.

My opinion:

The specification mining system can generate an automaton from running traces of programs using the specification, which depicts the behavior of the specification. The idea is good. However, to make it applicable to real life applications, the process needs to reduce these dependences on human interaction:

1. The program needs to be instrumented to get the traces. This job is not always trivial or simple.
2. The seeds need to be defined beforehand using human knowledge.
3. The maximum size of the scenario needs to be defined beforehand and to be tuned to get good results.
4. The STM need to be specified by human experts.
5. The results need to be examined by human experts.