Isolating Cause-Effect Chains from Computer Programs

Andreas Zeller

Presented by

Brian Russell

The Basic Idea

- Program execution considered as a sequence of state transitions.
- Sequence of states for correct execution will differ from sequence for faulty execution.
- Differences between state transitions provides valuable information.
- How to determine the difference?

Isolating Relevant Input

- Separate input files: one for correct execution, one for faulty execution.
- Input files need not be similar.
- Goal is to find minimal differences between “correct” and “fault causing” inputs.
- Method: divide and conquer …

Dividing and Conquering

- Automated isolation of failure inducing input.
- Basis is a sequence of atomic differences and an automated test function.
- Differences in execution runs becomes differences in program input.

Determining Precise Cause of Failure

- Define initial sets of atomic differences for correct and faulty outcomes.
- Define tests that determines correct/faulty result or ? result if program behaves unexpectedly.
- Iterative runs produce new sets of atomic differences that are more precise.
- When “correct” and “fault inducing” sets are close enough (1-minimal), stop.

Determining Precise Cause of Failure (continued)

- Increasingly precise cause requires intelligent modification of input.
- Automated delta debugging postulates “smart” splitter function to modify input.
- “Smart” splitting functions are not trivial as compiler example shows …
- Also assumes existence of automated test
- Input decomposition program in 10 minutes?

Claims

- Number of tests grows with number of unresolved test outcomes.
- Worst case is quadratic in size of fault inducing error atoms (How is this true?)
- Worst case does not occur in practice.
- Is exponential behavior more likely worst case?

Isolating Relevant Program States

- Program execution as series of states.
- Program state is defined in terms of variables and their values.
- Differences in program input cause differences in program states.
- Problem: even minimized changes in input can result in large changes in program execution state.

Isolating Relevant Program States

- Requires a source-level debugger.
- Map 1-minimal change in input to corresponding program source where state transition starts to deviate.
- Examine variables with delta debugging to find minimal set of variables involved in transition to fault related states.
- Set involved variables when necessary.

Memory Graphs

- Problem: exact values of pointers not relevant – what they point to is significant.
- Solution: Memory graph nodes are values and variables are arcs. Root has all variables as children.
- All values of same type and location in same node.
- What if sets of variables differ?

Different Graphs? Common Subgraphs

- Largest common subgraph NP-complete in worst case, but most precise.
- Large common subgraphs – compare the graphs in parallel traversal. O(|V|+|E|).
- When to get largest or just large common subgraph? Unclear.
- Differences become deltas individually applied to alter program variable state.

HOWCOME

- Prototype program state extractor.
- Program state redefined: not just variables and values, now includes program counter and call stack history.
- Example magnitudes: 27000+ nodes, 42000+ nodes, reduced to one node of interest.
- Example times: 1.5 hours + 30 minutes.

An important point

- A cause can be determined automatically, but the fault remains in the eye of the beholder.
- The programmer still has to fix the code.

A Curious Argument

Does the programmer have to narrow down the point of transition to a faulty state?

Maybe not:

- Increase granularity of cause-effect chain for more precision.
- Isolate cause transitions automatically.
- Use heuristics to focus on possible relevant events.

Surprising Results

- Changing variables to meaningless values is not entirely haphazard.
- If similar behaviors ensue, changed variable is irrelevant.
- Good code is easier to deal with than spaghetti (Duh!).
- Program state is easy to decompose, program input decomposition is hard.

Admitted Weaknesses

- Requirement of “correct” and “fault-inducing” inputs and program runs.
- Isolated causes may be only indirectly informative.
- Only one cause out of many may be isolated.
- A large difference may not always be narrowed.

Observed Weaknesses

- Could multiple problems preclude narrowing to any cause?
- Program state definition is inconsistent and probably incomplete.
- Postulates use of nontrivial “smart” splitter of input.
- Automated state extraction takes a long time.

Related Work - Slicing

- All program statements that could influence the variable values in a source statement of interest.
- Static slicing is independent of execution behavior.
- Dynamic slicing is specific to a program run and is more precise.
- Slices may still be too large to manage.

And Dicing …

- The difference between dynamic slices.
- Isolates effects under different conditions.
- Also dynamic invariants, which are dynamic checks on program execution behavior against an invariant model.
- Making the invariant model is the challenge.

… And Julienne Fries …

- “Automated” debugging of Prolog programs.
- Systematic queries about the subclauses.
- Who cares about Prolog?

Conclusions and Contributions

- Debugging as program states.
- Proof of concept: Fully automated means of narrowing down program states and runs to those relevant to a given state.